Improving Usability of Mobile Applications Through Speculation and Distraction Minimization

by

Kyungmin Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Kyungmin Lee

kyminlee@umich.edu

ORCID iD: 0000-0002-8055-7639

For Sara

ACKNOWLEDGEMENTS

There are so many people to extend my appreciation for making this dissertation possible. First of all, I would like to thank my co-advisors, Jason Flinn and Brian Noble for providing relentless support, wisdom, encouragement, and patience. I know I wasn't the easiest student to advise and I truly believe that I would not have completed my Ph.D. if they were not my advisors. I thank their tremendous effort in always guiding me to become a better researcher. I would like to especially thank Jason Flinn again for understanding me and pushing me to finish my Ph.D. He has consistently provided numerous meticulous feedback to make this dissertation possible. I am always amazed by his devotion to research and his students. Jason, I am honored to say that I was your Ph.D. student.

I thank David Chu for extending his mentorship from the summer internship at Microsoft Research and serving as a dissertation committee member. I also thank my other committee members, Tawanna Dilahunt, and Z. Morley Mao for their valuable feedback. Their input has made this dissertation more thorough and complete.

I thank all of the former and current students of our group for helping whenever I was in need. I owe so much to Mona Attariyan, Kaushik Veeraraghavan, Dan Peek, Ya-Yunn Su, Brett Higgins, and Minkyong Kim for sharing their wisdom as academic older-siblings and always looking out for me. All of you have made my life so much easier whenever I was looking for an internship or a full-time position, and always gave the right advise that I needed. I want to especially thank my undergraduate advisor and academic older-sibling, Landon Cox, to continuously guide me even after my undergraduate years. I still recall vividly multiple conversations that I had with him about whether to pursue a Ph.D. degree or not. I am always grateful for his honest advice.

I thank my drinking buddy of Ph.D. years and a best friend, Michael Chow. My grueling Ph.D. years were certainly more bearable because of his friendship. From listening to my rant on miscellaneous topics and grammar checking everything I write/speak to dog-sitting my dog whenever I was out of town, I cannot thank him enough for everything he has done for me. I apologize to him for not being a great friend to him as he was to me. I thank David Devecsery for always coming to my practice talks and providing valuable feedback to improve the talk. I also thank HyunJoong Joseph Lee for scanning and e-mailing dissertation edits to me multiple times while I was working on it remotely. Without his effort, this dissertation probably took a lot longer to complete.

Lastly, but definitely not the least, I thank my wife, Sara, for supporting me throughout my Ph.D. My life has changed so much ever since I met her on my third year of Ph.D. I can't thank her enough for always standing beside me no matter what had happened. She has given me the right advise and smile whenever I needed one. Sara, marrying you was the best thing that ever happened to me and I am excited about our next chapters of life. I love you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Improving Usability of Mobile Applications Through Speculation and Distraction Minimization

by

Kyungmin Lee

Chair: Jason Flinn and Brian Noble

We live in a world where mobile computing systems are increasingly integrated with our day-to-day activities. People use mobile applications virtually everywhere they go, executing them on mobile devices such as smartphones, tablets, and smartwatches. People commonly interact with mobile applications while performing other primary tasks such as walking and driving (e.g., using turn-by-turn directions while driving a car). Unfortunately, as an application becomes more mobile, it can experience resource scarcity (e.g., poor wireless connectivity) that is atypical in a traditional desktop environment. When critical resources become scarce, the usability of the mobile application deteriorates significantly.

In this dissertation, I create system support that enables users to interact smoothly with mobile applications when wireless network connectivity is poor and when the user's attention is limited. First, I show that speculative execution can mitigate user-perceived delays in application responsiveness caused by high-latency wireless network connectivity. I focus on cloud-based gaming, because the smooth usability of such application is highly dependent on low latency. User studies have shown that players are sensitive to as little as 60 ms of additional latency and are aggravated at latencies in excess of 100ms [12, 27, 90]. For cloud-based gaming, which relies on powerful servers to generate high-graphics quality gaming content, a slow network frustrates the user, who must wait a long time to see input actions reflected in the game. I show that by predicting the user's future gaming inputs and by performing visual misprediction compensation at the client, cloud-based gaming can maintain good usability even with 120 ms of network latency.

Next, I show that the usability of mobile applications in an attention-limited environment (i.e., driving a vehicle) can be improved by automatically checking whether interfaces meet best-practice guidelines and by adding attention-aware scheduling of application interactions. When a user is driving, any application that demands too much attention is an unsafe distraction. I first develop a model checker that systematically explores all reachable screens for an application and determines whether the application conforms to best-practice vehicular UI guidelines. I find that even well-known vehicular applications (e.g., Google Maps and TomTom) can often demand too much of the driver's attention. Next, I consider the case where applications run in the background and initiate interactions with the driver. I show that by quantifying the driver's available attention and the attention demand of an interaction, real-time scheduling can be used to prevent attention overload in varying driving conditions.

CHAPTER I

Introduction

Mobile applications are increasingly integrated with our daily lives. People check their e-mails while walking, use GPS-navigation applications to get turn-by-turn directions, and locate items in grocery stores with mobile applications. Unfortunately, as a computing device becomes more mobile, applications that run on it can become less usable due to conditions that are uncommon in desktop computing such as high network latency or competition for the user's attention from other activities.

In this dissertation, I create system support that enables users to interact smoothly with mobile applications when wireless network connectivity is poor and when the user's attention is limited. First, I focus on improving usability of mobile applications caused by poor wireless connectivity. Mobile applications often rely on remote cloud servers to generate content. When the mobile network is slow, application performance degrades significantly, leading to poor usability.

My work improves usability for cloud-based gaming applications, in which cloud servers execute the games on behalf of thin clients (e.g., mobile clients) that merely transmit UI input events and display output rendered by the servers [1, 2, 3, 22]. Smooth interactivity is especially critical in gaming. Studies have shown that game players are sensitive to as little as 60 ms of additional latency and are aggravated at latencies in excess of 100ms [12, 27, 90]. However, recent studies have found that the 95th percentile network latencies for 3G and LTE are over 600ms and 400ms, respectively [52, 53, 95].

Next, I show that the usability of mobile applications in an attention-limited environment (specifically, used when driving a vehicle) can be improved by automatically checking whether interfaces meet best-practice guidelines and by attention-aware scheduling of application interactions. When a vehicular application demands more attention than the user has available, it becomes unusable because it distracts from the primary focus of driving vehicle.

However, when well-designed, the same vehicular application can provide significant benefit to the user by assisting them in navigating, finding a parking spot, etc. Unfortunately, understanding these distraction factors is difficult. Current best practice for vehicular application deployment is to have experienced vehicular UI experts manually check applications and work with developers to improve them before they can be deployed on in-vehicle infotainment systems. Because such effort does not scale to large numbers of applications, there is a need for automated methods that can verify vehicular application interfaces with limited human effort.

Additionally, vehicular applications often run in the background and initiate interactions when they receive new information (e.g., a new text message). Currently, these interactions can happen at any time without considering the user's available attention or disabled completely when a vehicle is in motion. This binary allow all or deny all approach creates the problem of overloading the driving in challenging driving conditions and not delivering important information in less-challenging conditions. Therefore, there is need to schedule application-initiated interactions taking into account attention supply and demand, as well as the priority of the interaction.

It is my thesis that:

System support can improve mobile application usability in challenging conditions caused by both computing and non-computing resources. Speculative execution can mask the usability degradation caused by high network latency, a computing resource, in mobile cloud gaming. Automatically verifying whether vehicular applications follow best-practice UI guidelines and attention-aware scheduling of application-initiated interactions can minimize distraction in vehicular computing where available user attention, a non-computing resource, is scarce.

To validate this thesis, I have completed the following pieces of work.

Using Speculation to Enable Low-latency Continuous Interaction for Mobile Cloud Gaming: I have developed Outatime, which utilizes speculative execution and CUDA encoding/decoding to provide high game interactivity for network latency (RTT) as high as 120ms. Outatime's basic approach is to speculatively generate one or more *possible worlds,* one of which ought to correspond with the player's actual world RTT milliseconds in the future. To perform speculation, Outatime predicts user's future inputs based on user's current input. In the case where user's possible future input space is too big, Outatime performs state approximation to reduce the number of possible future inputs. Additionally, Outatime utilizes image-based rendering (IBR), a technique that transforms pre-rendered images from one viewpoint to another, to compensate for mispredictions and render the corrected state on a thin client. Chapter II discusses Outatime in more detail.

Verifying User Interface Properties for Vehicular Applications: I have implemented an automatic toolkit, called AMC that checks Android applications for vehicular UI guideline violations. When I used AMC on popular applications in the Android market, I discovered that vehicular applications were not substantially better at meeting best practice guidelines than non-vehicular applications. Chapter III discusses AMC in more detail.

Scheduling Application-Initiated Interactions in Vehicular Computing: I have developed Gremlin, which provides Android support for scheduling application-initiated interactions in a vehicular computing environment. Gremlin estimates the visual and cognitive attention demand of an interaction by recording a driver's past interactions and analyzing them offline. When an application initiates an interaction, Gremlin computes the driver's currently available visual and cognitive attention by considering vehicle speed, road curvature, and the driver's experience. Gremlin only allows the interaction to be initiated if the driver has enough attention to handle the interaction. When the driver does not have enough attention available, Gremlin defers scheduling the interaction until a more appropriate time (e.g., when the vehicle is stopped at a red light). I have evaluated Gremlin by comparing its decisions to those of a vehicle UI expert, and found that Gremlin closely approximates the expert's decisions. Chapter IV discusses Gremlin's design and contributions in more detail.

CHAPTER II

Attaining Low-Latency Continuous Interactivity over the Wide-Area with Speculation

In the first part of my thesis, I focus on gaming, since such applications require the highest interactivity with the user. More specifically, I focus on improving usability of cloud-based gaming application caused by poor wireless connectivity. Gaming is the most popular mobile activity, accounting for nearly a third of the time spent on mobile devices [38]. Recently, cloud gaming — where datacenter servers execute the games on behalf of thin clients that merely transmit UI input events and display output rendered by the servers — has emerged as an interesting alternative to traditional client-side execution. Cloud gaming offers several advantages. First, every client can enjoy the high-end graphics provided by powerful server GPUs. This is especially appealing for mobile devices such as basic laptops, phones, tablets, TVs and other displays lacking high-end GPUs. Second, cloud gaming eases developer pain. Platform compatibility headaches and vexing per-platform performance tuning — sources of much developer frustration [119, 106, 89] — are eliminated. Third, cloud gaming simplifies software deployment and management. Server management (e.g., for bug fixes, software updates, hardware upgrades, content additions, etc.) is far easier than modifying clients. Finally, players can select from a vast library of titles and instantly play any of them. Sony, Nvidia, Amazon and OnLive are among the providers that currently offer cloud gaming services [1, 2, 3, 22].

However, cloud gaming faces a key technical dilemma: how can players attain *real-time interactivity* in the face of wide-area latency? Real-time interactivity means client input events should be quickly reflected on the client display. User studies have shown that players are sensitive to as little as 60 ms latency, and are aggravated at latencies in excess of 100ms [27, 90, 12]. A further delay degradation from 150ms to 250ms lowers user engagement by 75% [21].

One way to address latency is to move servers closer to clients. Unfortunately, not only are decentralized edge servers more expensive to build and maintain, local spikes in demand cannot be routed to remote servers which further magnifies costs. Most importantly, high latencies are often attributed to the networks's last mile. Recent studies have found that the 95th percentile of network latencies for 3G, Wi-Fi and LTE are over 600ms, 300ms and 400ms, respectively [53, 52, 95]. In fact, even well-established residential wired last mile links tend to suffer from latencies in excess of 100ms when under load [101, 7]. Unlike non-interactive video streaming, buffering is not possible for interactive gaming.

Instead, I mitigate wide-area latency via speculative execution. My system, *Outatime*,[1] delivers real-time gaming interactivity as fast as — and in some cases, even faster than — traditional local client-side execution, despite latencies up to 120ms.

Outatime's basic approach is to employ speculative execution to render multiple possible frame outputs which could occur RTT milliseconds in the future. While the fundamentals of speculative execution are well-understood, the *dynamism* and *sensitivity* of graphically intensive twitch-based interaction makes for stringent demands on speculation. Dynamism leads to rapid state space explosion. Sensitivity means users are able to (visually) identify incorrect states. Outatime employs the following techniques to manage speculative state in the face of dynamism and sensitivity.

[1] *Outatime* : License plates of a car capable of time travel.

Future State Prediction: Given the user's historical tendencies and recent behavior, I show that some categories of user actions are highly predictable. I develop a Markov-based prediction model that examines recent user input to forecast expected future input. I use two techniques to improve prediction quality: supersampling of input events, and Kalman filtering to improve users' perception of smoothness.

State Approximation: For some types of mispredictions, Outatime approximates the correct state by applying *error compensation* on the (mis)predicted frame. The resulting frame is very close to what the client ought to see. The misprediction compensation uses *image-based rendering (IBR)*, a technique that transforms pre-rendered images from one viewpoint to a different viewpoint using only a small amount of additional 3D metadata.

Certain user inputs (*e.g.,* firing a gun) cannot be easily predicted. For these, I use parallel speculative executions to explore multiple outcomes. However, the set of all possible states over long RTTs can be very large. To address this, I approximate the state space by subsampling it, and time shifting actual event streams to match event streams that are "close enough" *i.e.,* below players' sensitivity thresholds. These techniques greatly reduces the state space with minimal impact on the quality of interaction, thereby permitting speculation within a reasonable deadline.

State Checkpoint and Rollback: Core to Outatime is a real-time process checkpoint and rollback service. Checkpoint and rollback prevents mispredictions from propagating forward. I develop a hybrid memory-page based and object-based checkpoint mechanism that is fast and well-suited for graphically-intensive real-time simulations like games.

State Compression for Saving Bandwidth: Speculation's latency reduction benefits come at the cost of higher bandwidth utilization. To reduce this overhead, I develop a video encoding scheme which provides a 40% bitrate reduction over standard encoding by taking advantage of the visual coherence of speculated frames. The final bitrate overhead of speculation depends on the RTT. It is 1.9× for 120ms.

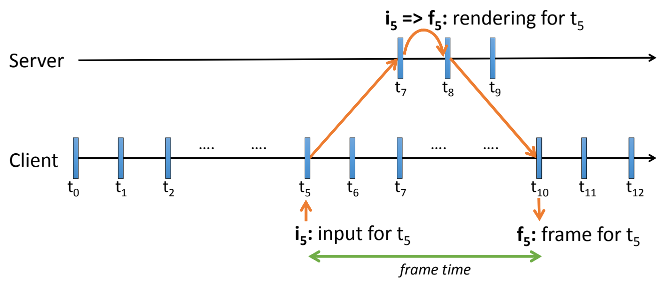
To illustrate Outatime in the context of fast interaction, I evaluate Outatime's prediction techniques using two action games where even small latencies are disadvantageous. Doom 3 is a twitch-based first person shooter where responsiveness is paramount. Fable 3 is a role playing game with frequent fast action combat. Both are high-quality, commercially-released games, and are very similar to phone and tablet games in the first person shooter and role playing genres, respectively.

Through interactive gamer testing, I found that even experienced players perceived only minor differences in responsiveness on Outatime when operating at up to 120ms RTT when compared head-to-head to a system with no latency. Latencies up to 250ms RTT were even acceptable for less experienced players. Moreover, unlike in standard cloud gaming systems, Outatime players' in-game skills performance did not drop off as RTT increased up to 120ms. Overall, player surveys scored Outatime gameplay highly. I have also deployed Outatime to the general public on limited occasions and received positive reception [66, 105, 40, 31]. While I have focused the evaluation on desktop and laptop clients experiencing high network latency, techniques are broadly applicable to phone and tablet clients as well. Outatime's only client prerequisites are video decode and modest GPU capabilities which are standard in today's devices.

2.1 Background & Impact of Latency

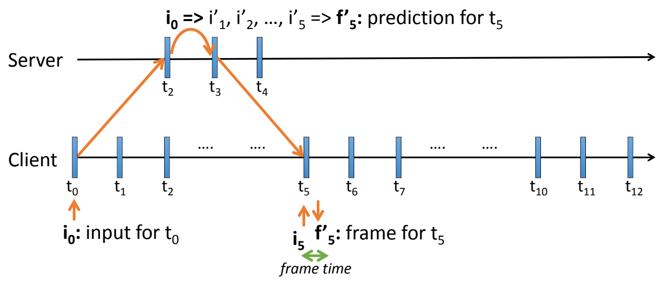
The vast majority of game applications are structured around the *game loop,* a repetitive execution of the following stages: 1) read user input; 2) update game state; and 3) render and display frame. Each iteration of this loop is a logical tick of the game clock and corresponds to 32ms of wall-clock time for an effective frame rate of 30 frames per second (fps).[2] The time taken for one iteration of the game loop is the *frame time.* Frame time is a key metric for assessing interactivity since it corresponds

[2] $\frac{1}{30fps} \approx 32$ms for mathematical convenience.



i_5 => f_5: rendering for t_5
Server
Client
t_7 t_8 t_9
t_0 t_1 t_2 t_5 t_6 t_7 t_{10} t_{11} t_{12}
i_5: input for t_5
f_5: frame for t_5
frame time

(a) Standard cloud gaming: Frame time depends on net latency.



i_0 => i'_1, i'_2, ..., i'_5 => f'_5: prediction for t_5
Server
Client
t_2 t_3 t_4
t_0 t_1 t_2 t_5 t_6 t_7 t_{10} t_{11} t_{12}
i_0: input for t_0
i_5
f'_5: frame for t_5
frame time

(b) Outatime: Frame time is negligible.

Figure 2.1: Comparison of frame delivery time lines. RTT= 4 ticks, server processing time = 1 tick.

to the delay between a user's input and observed output.

Network latency has an acute effect on interaction for cloud gaming. In standard cloud gaming, the frame time must include the additional overhead of the network RTT, as illustrated in Figure 2.1a. Let time be discretized into 32ms clock ticks, and let the RTT be 4 ticks (128ms). At t_5, the client reads user input i_5 and transmits it to the server. At t_7, the server receives the input, updates the game state, and renders the frame, f_5. At t_8, the server transmits the output frame to the client, which receives it at t_{10}. Note that the frame time incurs the full RTT overhead. In this example, an RTT of 128ms results in a frame time of 160 ms.

2.2 Goals and System Architecture

For Outatime, responsiveness is paramount; Outatime's goal is to consistently deliver low frame times (< 32ms) at high frame rate (> 30fps) even in the face of long RTTs. In exchange, I am willing to transmit a higher volume of data and potentially even introduce (small and ephemeral) visual artifacts, ideally sufficiently minor that most players rarely notice.

The basic principle underlying Outatime is to speculatively generate possible output frames and transmit them to the client a full RTT ahead of the client's actual corresponding input. As shown in Figure 2.1b, the client sends input as before; at t_0, the client sends the input i_0 which happens to be the input generated more than one RTT interval prior to t_5. The server receives i_0 at t_2, computes a sequence of probable future input up to one RTT later as $i'_1, i'_2, ..., i'_5$ (I use $'$ to denote speculation), renders its respective frame f'_5, and sends these to the client. Upon reception at the client at time t_5, the client verifies that its actual input sequence recorded during the elapsed interval matches the server's predicted sequence: $i_1 = i'_1, i_2 = i'_2, ..., i_5 = i'_5$. If the input sequences match, then the client can safely display f'_5 without modification because I ensure that the game output is deterministic for a given input [25]. If the input sequence differs, the client approximates the actual state by applying *error compensation* to f'_5 and displays a corrected frame. I describe error compensation in detail in Section 2.3.5. Unlike in standard cloud gaming where clients wait more than one RTT for a response, Outatime immediately delivers response frames to the client after the corresponding input.

Speculation performance in Outatime depends upon being able to accurately predict future input and generate its corresponding output frames. Outatime does this by identifying two main classes of game input, and building speculation mechanisms for each, as illustrated in Figure 2.2. The first class, *navigation*, consists of input events that control view (rotation) and movement (translation) and modify the player's field of view. Navigation inputs tend to exhibit continuity over short time windows, and therefore Outatime makes effective predictions for navigation.

The second class, *impulse*, consists of events that are inherently sporadic such as

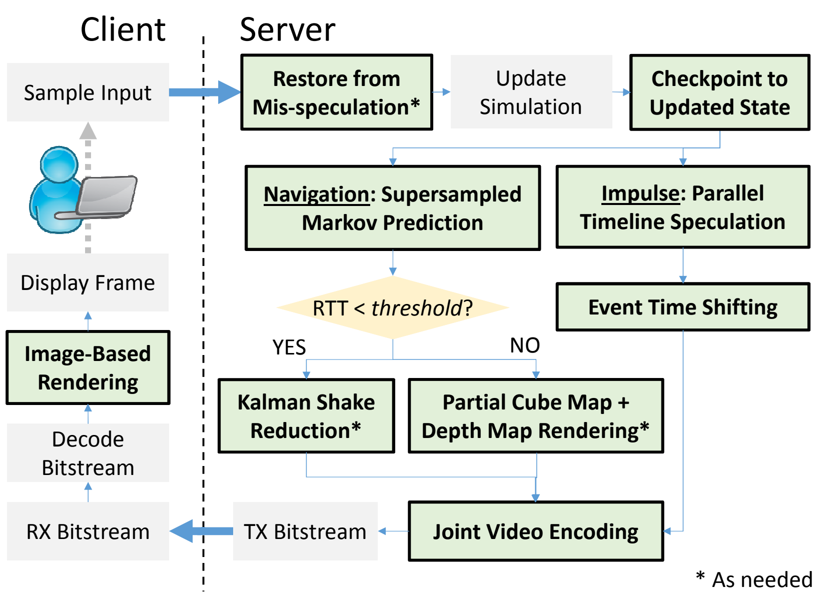


Figure 2.2: The Outatime Architecture. Bold boxes represent the main areas of technical focus.

firing a weapon or activating an object, yet are fundamental to the player's perception of responsiveness. For example, in first person shooters, instantaneous weapon firing is core to gameplay. Unlike navigation inputs, the sporadic nature of impulse events makes them less amenable to prediction. Instead, Outatime generates parallel speculations for multiple possible future impulse time lines. To tame state space explosion, Outatime subsamples the state space and time shifts impulse events to the closest speculated timeline. This enables Outatime to provide the player the perception that impulse is handled instantaneously. Besides navigation and impulse, I classify other input that is slow relative to RTT as *delay tolerant*. One typical example of delay tolerant input is activating the heads-up display. Delay tolerant input is not subject to speculation, and I discuss how it is handled in Section 2.4.

Figure 2.2 also shows how Outatime's server and client are equipped to deal with speculations that are occasionally wrong. The server continually checkpoints valid state and restores from (mis-)speculated state at 30 fps. The client executes IBR — a very basic graphics procedure — to compensate for navigation mispredictions when they occur. Otherwise, Outatime, like standard cloud gaming systems, makes minimal assumptions about client capabilities. Namely, the client should be able to perform standard operations such as decode a bitstream, display frames and transmit standard input such as button, mouse, keyboard and touch events. In contrast, high-end games that run solely on a client device can demand much more powerful CPU and GPU processing, as I show in Section 2.9.

2.3 Speculation for Navigation

Navigation speculation entails predicting a sequence of future navigation input events at discrete time steps. Hence, I use a discrete time Markov chain for navigation inference. I first describe how I applied the Markov model to input prediction, and my use of supersampling to improve the inference accuracy. Next, I refine the prediction in one of two ways, depending on the severity of the expected error. I determine the expected error as a function of RTT from offline training. When errors are sufficiently low (typically corresponding to RTT$<$ 40ms), I apply an additional Kalman filter to reduce video “shake”). Otherwise, I use misprediction compensation on the client to post-process the frame rendered by the server.

2.3.1 Basic Markov Prediction

I construct a Markov model for navigation. Time is quantized, with each discrete interval representing a game tick. Let the random variable navigation vector N_t represent the change in 3-D translation and rotation at time t:

$$N_t = \{\delta_{x,t}, \delta_{y,t}, \delta_{z,t}, \theta_{x,t}, \theta_{y,t}, \theta_{z,t}\}$$

Each component above is quantized. Let n_t represent an actual empirical navigation vector received from the client. The state estimation problem is to find the maximum likelihood estimator $\hat{N}_{t+\lambda}$ where λ is the RTT.

Using the Markov model, the probability distribution of the navigation vector at the next time step is dependent only upon the navigation vector from the current time step: $p(N_{t+1}|N_t)$. I predict the most likely navigation vector $\hat{N}_{t+1}$ at the next time step as:

$$\begin{aligned}\hat{N}_{t+1} &= E[p(N_{t+1}|N_t = n_t)] \\ &= argmax_{N_{t+1}} p(N_{t+1}|N_t = n_t)\end{aligned}$$

where $N_t = n_t$ indicates that the current time step has been assigned a fixed value by sampling the actual user input n_t. In many cases, the RTT is longer than a single time step (32ms). To handle this case, I predict the most likely value after one RTT as:

$$\hat{N}_{t+\lambda} = argmax_{N_{t+\lambda}} p(N_{t+1}|N_t = n_t) \prod_{i=1..\lambda-1} p(N_{t+i+1}|N_{t+i})$$

where λ represents the RTT latency expressed in units of clock ticks.

My results indicate that the Markov assumption holds up well in practice: namely, N_{t+1} is memoryless (i.e., independent of the past given N_t). In fact, additional history in the form of longer Markov chains did not show a measurable benefit in terms of prediction accuracy. Rather than constructing a single model for the entire navigation vector, instead I treat each component of the vector N independently, and construct six separate models. The benefit of this approach is that less training is required when estimating $\hat{N}$, and I observed that this assumption of treating the vector components independently does not hurt prediction accuracy. Below in 2.3.3, I discuss the issue of training in more detail.

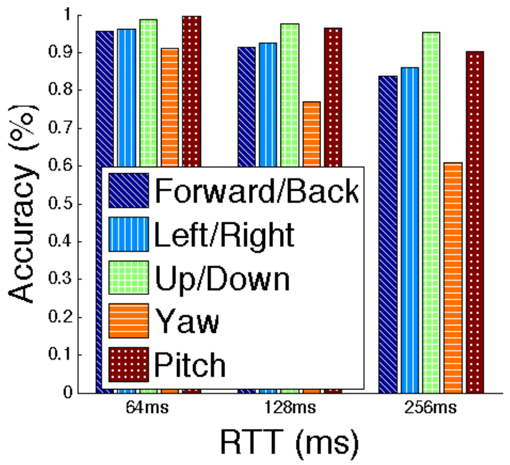
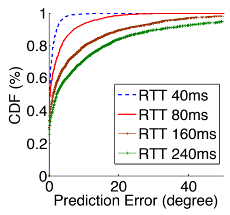


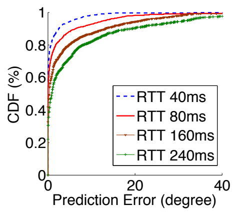
Figure 2.3: Doom 3 Navigation Prediction Summary. Roll (θ_z) is not an input in Doom 3 and need not be predicted.

2.3.2 Supersampling

I further refine the navigation predictions by *supersampling*: sampling input at a rate that is faster than the game's usage of the input. Supersampling helps with prediction accuracy because it lowers sampling noise. To construct a supersampled Markov model, I first poll the input device at the fastest rate possible. This rate is dependent on the specific input device. It is at least 100Hz for touch digitizers and at least 125Hz for standard mice. With a 32ms clock tick, I can often capture at least four samples per tick. I then build the Markov model as before. The inference is similar to the equation above, with the main difference being the production operator incrementing by $i \overset{\pm}{=} 0.25$. A summary of navigation prediction accuracy from the user study described in 2.9 is shown in Figure 2.3. Most dimensions of rotational and translational displacement exhibit little performance degradation with longer RTTs. Yaw (θ_x), player's horizontal view angle, exhibits the most error, and I show its performance in detail in Figure 2.4 for user traces collected from both Doom 3 and Fable 3 at various RTTs from 40ms to 240ms. Doom 3 exhibits greater error than Fable 3 due to its more frenetic gameplay. Based on subjective assessment,



(a) Doom 3



(b) Fable 3

Figure 2.4: Prediction for Yaw (θ_x), the navigation component with the highest variance. Error under 4° is imperceptible.

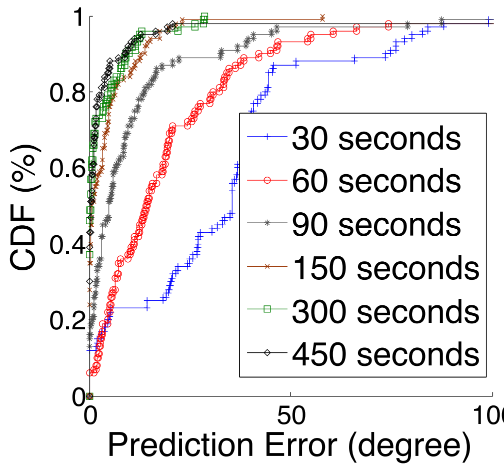


Figure 2.5: Error Decreases with More Observation Time. Data is for Fable 3 at RTT= 160ms.

prediction error below 4° is under the threshold at which output frame differences are perceivable.

Based on these results, I make two observations. First, for RTT $\leq$ 40ms (where 98% and 93% of errors are less than 4° for Doom 3 and Fable 3 respectively), errors are sufficiently minor and infrequent. Note that the client can detect the magnitude of the error (because it knows the ground truth), and drop any frames with excessive error. A frame rate drop from 30fps to $30 \times 0.95 = 28.5$fps is unlikely to affect most players' perceptions. For RTT $>$ 40ms, I require additional misprediction compensation mechanisms. Before discussing both of these cases in turn, I first address the question of how much training is needed for successful application of the predictive model.

2.3.3 Bootstrap Time

Construction of a reasonable Markov Model requires sufficient training data collected during an observation period. Figure 2.5 shows that prediction error improves as observation time increases from 30 seconds to 300 seconds, after which the prediction error distribution remains stable. Somewhat surprisingly, having test players different from training players only marginally impacts prediction performance as long as test and train players are of similar skill level. Therefore, I chose to use a single model per coarse-grained skill level (novice or experienced) which is agnostic of the player.

2.3.4 Shake Reduction with Kalman Filtering

While the Markov model yields high prediction accuracy for RTT$<$ 40ms, minor mispredictions can introduce a distracting visual effect that I describe as video shake. As a simple example, consider a single dimension of input such as yaw. The ground truth over three frames may be that the yaw remains unchanged, but the prediction error might be $+2^\circ, -3^\circ, +3^\circ$. Unfortunately, the user would perceive a shaking effect because the frames would jump by 5° in one direction, and then 6° in another. From the experience with early prototypes, the manifested shakiness was sufficiently noticeable so as to reduce playability.

I apply a Kalman filter [60] in order to compensate for video shake. The filter's advantage is that it weighs estimates in proportion to sample noise and prediction error. Conceptually, when errors in past predictions are low relative to sample noise, predictions are given greater weight for state update. Conversely, when measurement noise is low, samples make greater contribution to the new state. One interesting filter modification I make is that I extend the filter to support error accumulation over variable RTT time steps; samples are weighed against an RTT's worth of prediction error.

2.3.5 Misprediction Compensation with Image-based Rendering

When RTT $>$ 40ms, a noticeable fraction of navigation input is mispredicted, resulting in users perceiving lack of motor control. The goal in misprediction com-

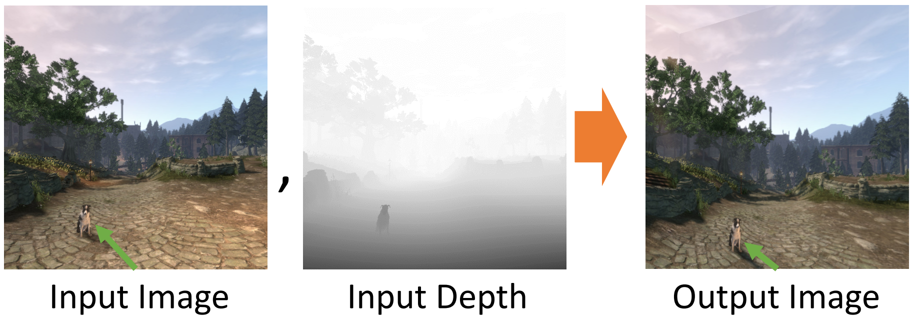


Figure 2.6: Image-based Rendering Example w/ Fable 3. Forward translation and leftward rotation is applied. The dog (indicated by green arrow) is closer and toward the center after IBR.

pensation is for the server to generate auxiliary view data f^{Δ} alongside its predicted frame f' such that the client can reconstruct a frame f'' that is a much better approximation of the desired frame f than f'.

2.3.5.1 Image-based Rendering

I compensate for mispredictions with *image-based rendering (IBR)*. IBR is a means to derive novel camera viewpoints from fixed initial camera viewpoints. The specific implementation of IBR that I use operates by having initial cameras capture depth information (f^{Δ}) in addition to 2D RGB color information (f'). It then applies a warp to create a new 2D image (f'') from f' and f^{Δ} [102]. Figure 2.6 illustrates an example whereby an original image and its depth information is used to generate a new image from a novel viewpoint. Note that the new image is both translated and rotated with respect to the original, and contains some visual artifacts in the form of blurred pixels and low resolution areas when IBR is inaccurate.

To enable IBR, two requirements must be satisfied. First, the depth information must accurately reflect the 3D scene. Fortunately, the graphics pipeline's z-buffer precisely contains per-pixel depth information and is already a byproduct of standard rendering. Second, the original 2D scene must be sufficiently large so as to ensure that any new view is bounded within the original. To handle this case, instead of

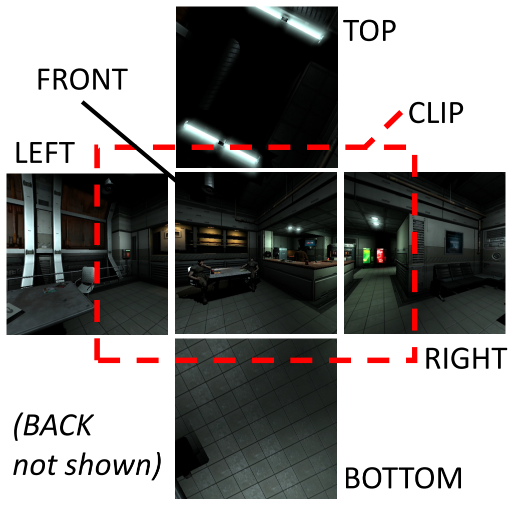


Figure 2.7: Cube Map Example w/ Doom 3. Clip region shown.

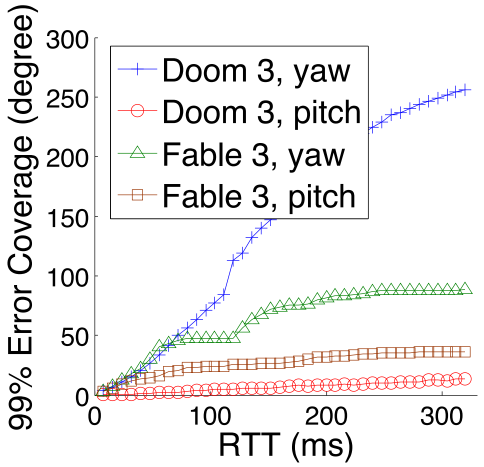


Figure 2.8: Angular coverage of 99% of prediction errors is much less than 360° even for high RTT.

rendering a normal 2D image by default, I render a cube map [35] centered at the player's position. As shown in Figure 2.7, the cube map draws a panoramic 360° image on the six sides of a cube. In this way, the cube map ensures that any IBR image is within its bounds.

Unfortunately, naïve use of the depth map and cube map can lead to significant overhead. The cube map's six faces are approximately[3] six times the size of the original image. The z-buffer is the same resolution as the original image, and depth information is needed for every cube face. Taken together, the total overhead is nominally 12×. This cost is incurred at multiple points in the system where data size is the main determinant of resource utilization, such as server rendering, encoding, bandwidth and decoding. I use the following technique to reduce this overhead.

2.3.5.2 Clipped Cube Map

I observe that it is unlikely that the player's true view will diverge egregiously from the most likely predicted view; transmitting a cube map that can compensate

[3]The original image is not square but rather 16:9 or 4:3.

for errors in 360° is gratuitous. Therefore, I render a *clipped cube map* rather than a full cube map. The percentage of clipping depends on the expected variance of the prediction error. If the variance is high, then I render more of the cube. On the other hand, if the prediction variance is low, I render less of the cube. The dotted line in Figure 2.7 marks the clip region for an example rendering.

In order to size the clip, I define a cut plane c such that the clipped cube bounds the output image with probability $1 - \epsilon$. The cut plane then is a function of the variance of the prediction, and hence the partial cube map approaches a full cube when player movement exhibits high variance over the subject RTT horizon. To calculate c, I choose not a single predicted Markov state, but rather a set $\mathcal{N}$ of k states such that the set covers $1 - \epsilon$ of the expected probability density:

$$\mathcal{N} = \{n^i_{t+1} | \sum_{i=1..k} p(N_{t+1} = n^i_{t+1} | N_t = n_t) \geq 1 - \epsilon\}$$

The clipped cube map then only needs to cover the range represented by the states in $\mathcal{N}$. For a single dimension such as yaw, the range is then simply the largest distance difference, and the cut plane along the yaw axis is defined as follows:

$$c_{yaw} = \max_{n^i_{t+1} \in \mathcal{N}} yaw(n^i_{t+1}) - \min_{n^j_{t+1} \in \mathcal{N}} yaw(n^j_{t+1})$$

This suffices to cover $1 - \epsilon$ of the probable yaw states.

In practice, error ranges are significantly less than 360° and therefore the size of the cube map can be substantially reduced. Figure 2.8 shows the distribution of c_{yaw} and c_{pitch} in Fable 3 and Doom 3 for $\epsilon = 0.01$, meaning that 99% of mispredictions are compensated. Doom 3's pitch range, player's vertical view angle, is very narrow (because players hardly look up or down), and both Fable 3's yaw and pitch ranges are modest at under 80° even for RTT $\geq 300ms$. Even for Doom 3's pronounced yaw range, only 225° of coverage is needed at 250 ms. The clip parameters are also

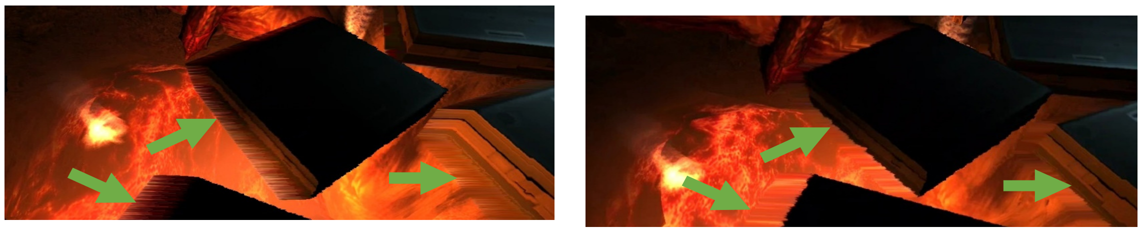

(a) Visible Smears (b) Patched Smears

Figure 2.9: Misprediction's visual artifacts appear as smears which I mitigate.

applied to the depth map in order to similarly reduce its size.

In theory, compounding translation error on top of rotation error can further expand the clip region. It turns out that translation accuracy (see Figure 2.3) is sufficiently high to obviate consideration of accumulated translation error for the purposes of clipping.

2.3.5.3 Patching Visual Artifacts

Lastly, I add a technique to mitigate visual artifacts. These occasionally appear when the navigation prediction is wrong and there is significant depth disparity between foreground and background objects. For example, in Figure 2.9a, the floating stones are much closer to the camera than the lava in the background. As a result, IBR reveals "blind spots" behind the stones as indicated by the green arrows, which are manifest as visual "smears".

The blind spot patching technique mitigates this problem, as seen in Figure 2.9b. As part of IBR, I extrude the object borders in the depth map with lightweight image processing which propagates depth values from foreground borders onto the background. However, I keep the color buffer the same. This way blind spots will exhibit less depth disparity and will share adjacent colors with the background, appearing as more reasonable extensions of the background. The results are more pleasing scenes especially when near and far objects are intermingled.

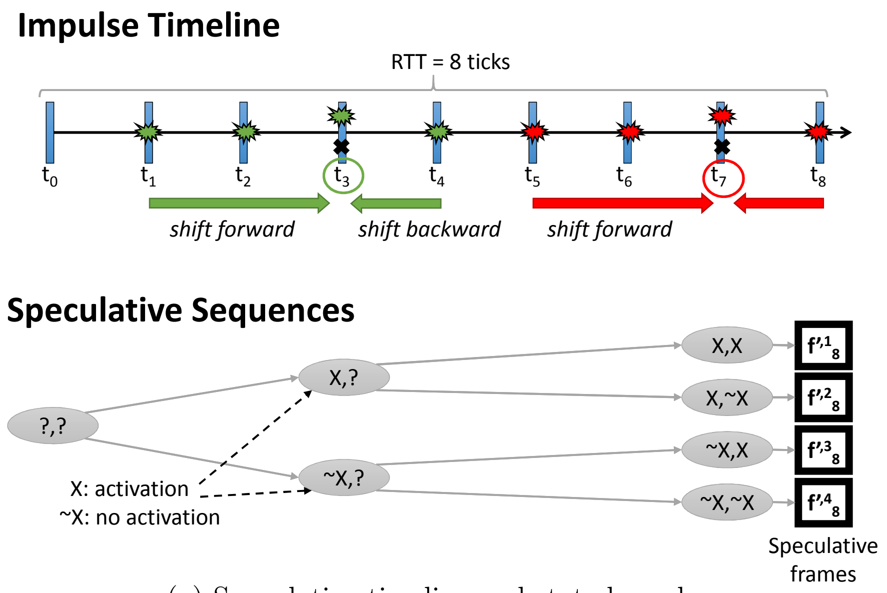
2.4 Speculation for Impulse Events

The prototypical impulse events are FIRE for first person shooters, and INTERACT (with other characters or objects) for role playing games. I define an impulse event as being *registered* when its corresponding user input is *activated*. For example, a user's button activation may register a FIRE event.

The objective for impulse speculation is to respond quickly to player's impulse input while avoiding any visual inconsistencies. For example, in a first person shooter, weapons should fire quickly when triggered, and enemies should not reappear shortly after dying. The latter type of visual (and semantic) inconsistency is disconcerting to players, yet may occur when mispredictions occur in a prediction-based approach. Therefore, I employ a speculation technique for impulse that differs substantially from navigation speculation. Rather than attempt to predict impulse events, instead I explore multiple outcomes in parallel.

An overview of Outatime's impulse speculation is as follows. The server creates a *speculative input sequence* for all possible event sequences that may occur within one RTT, executes each sequence, renders the final frame of each sequence, and sends the set of speculative input sequences and frame pairs to the client. Upon reception, the client chooses the event sequence that matches the events that actually transpired, and displays its corresponding frame.

As RTT increases, the number of possible sequences grows exponentially. Consider an RTT of 256ms, which is 8 clock ticks. An activation may lead to an event registration at any of the 8 ticks, leading to an overwhelming 2^8 possible sequences. In general, 2^λ sequences are possible for an RTT of λ ticks. I use two state approximations to tame state space explosion: subsampling and time-shifting.



(a) Speculative timeline and state branches

(b) ∼X, ∼X

(c) ∼X, X

(d) X,∼X

(e) X,X

Figure 2.10: Subsampling and time-shifting impulse events allows the server to bound speculation to a maximum of four sequences even for RTT= 256ms. Screenshots (b) – (e) show speculative frames corresponding to four activation sequences of weapon fire and no fire.

2.4.1 Subsampling

I reduce the number of possible sequences by only permitting activations at the subsampling periodicity σ which is a periodicity greater than one clock tick. The benefit is that the state space is reduced to $2^{\frac{\lambda}{\sigma}}$. The drawback is that subsampling alone would cause activations not falling on the sampling periodicity to be lost, which would appear counter-intuitive to users.

2.4.2 Time-Shifting

To address the shortcomings of subsampling, *time-shifting* causes activations to be registered either earlier or later in time in order to align them with the nearest subsampled tick. Time shifting to an earlier time is feasible using speculation because the shift occurs on a speculative sequence at the server — not an actual sequence that has already been committed by the client. Put another way, as long as the client has not yet displayed the output frame at a particular tick, it is always safe to shift an event backwards to that tick.

Specifically, for any integer k, an activation issued between $t_{\sigma*k-\frac{\sigma}{2}}$ and $t_{\sigma*k-1}$ is deferred until $t_{\sigma*k}$. An activation issued between $t_{\sigma*k+1}$ and $t_{\sigma*k+\frac{\sigma}{2}-1}$ is treated as if it had arrived earlier in time at $t_{\sigma*k}$. Figure 2.10a illustrates combined subsampling and time-shifting, where the activations that occur at t_1 through t_2 are shifted later to t_3 and activations that occur at t_4 are shifted earlier to t_3. The corresponding state tree in Figure 2.10a shows the possible event sequences and four resulting speculative frames, f'^1_8, f'^2_8, f'^3_8 and f'^4_8.

Note that it is not necessary to handle activations at t_0 within the illustrated 8 tick window because speculations that started at earlier clock ticks (e.g. at t_{-1}) would have covered them.

The ability to time-shift both forward and backward allows us to further halve the subsampling rate to double σ without impacting player perception. Using 60ms as the threshold of player perception [90, 12], I note that time-shifting forward alone permits a subsampling period of $\sigma = 2$ (64ms) with an average shift of 32ms. With the added ability to time-shift backward as well, I can support a subsampling period of $\sigma = 4$ (128ms) yet still maintain an average shift of only 32ms. For $\sigma = 4$ and RTT$\leq$ 256ms, I generate a maximum of four speculative sequences as shown in Figure 2.10a. When RTT $>$ 256ms, I further lower the subsampling frequency sufficiently to ensure that I bound speculation to a maximum of four sequences. Specifically, $\sigma = \frac{\lambda}{2}$. While this can potentially result in users noticing the lowered sample rate, it allows us to cap the overhead of speculation.

2.4.3 Advanced Impulse Events

While binary impulse events are the most common, some games provide more options. For example, a Fable 3 player may cast a magic spell either directionally or unidirectionally which is a ternary impulse event due to mutual exclusion. Some first person shooters support primary and secondary fire modes (Doom 3 does not) which is also a ternary impulse event. With a ternary (or quaternary) impulse event, the state branching factor is three (or four) rather than two at every subsampling tick. With four parallel speculative sequences and a subsampling interval of $\sigma = 128$ms, Outatime is able to support RTT $\leq$ 128ms for ternary and quaternary impulse events without lowering the subsampling frequency.

2.4.4 Delay Tolerant Events

I classify any input event that is slow relative to RTT as *delay tolerant*. I use a practical observation to simplify handling of delay tolerant events. According to measurements on Fable 3 and Doom 3, delay tolerant events exhibited very high *cool down times* that exceeded 256ms. The cool down time is the period after an event is registered during which no other impulse events can be registered. For example, in Doom 3, weapon reloading takes anywhere from 1000ms to 2500ms during which time the weapon reload animation is shown. Iapon switching takes even longer. Fable 3 delay tolerant events have even higher cool down times. I take the approach that whenever a delay tolerant input is activated at the client, it is permissible to miss one full RTT of the event's consequences, as long as I can *compress time* after the RTT. The time compression procedure works as follows: for a delay tolerant event which displays τ frames worth of animation during its cool down (e.g. a weapon reload animation which takes τ frames), I may miss λ frames due to the RTT. During the remaining $\tau - \lambda$ frames, I choose to compress time by sampling $\tau - \lambda$ frames uniformly from the original animation sequence τ. The net effect is that delay tolerant event animations appear to play at fast speed. In return, I are assured that all events are properly processed because the delay tolerant event's cool down is greater than the RTT. For example, weapon switching or reloading immediately followed by firing is handled correctly.

2.5 Fast Checkpoint and Rollback

As with other systems that perform speculation [78, 116], Outatime uses checkpoint and restore to play forward a speculative sequence, and roll back the sequence if it turns out to be incorrect. In contrast to these previous systems, the continuous 30fps interactivity performance constraints are qualitatively much more demanding, and I highlight how I have managed these requirements.

Unique among speculation systems, I use a hybrid of page-level checkpointing and object-level checkpointing. This is because I need very fast checkpointing and restore; whereas page-level checkpointing is efficient when most objects need checkpointing, object-level checkpointing is higher performance when few objects need checkpointing. In general, it is only necessary to checkpoint *Simulation State Objects (SSOs)*: those objects which reproduce the world state. Checkpointing objects which have no bearing on the simulation, such as buffer contents, only increase runtime overhead. Therefore, I choose either object-level or page-level checkpointing based on the density of SSOs among co-located objects. I currently make the choice between page-level or object-level manually at the granularity of the binaries (executables and libraries), though it is also conceivable to do so automatically at the level of the compilation units.

For page-level checkpointing, I intercept calls to the default `libc` memory allocator with a version that implements page-level copy-on-write. At the start of a speculation (at every clock tick for navigation and at each σ clock ticks for impulse), the allocator marks all pages read-only. When a page fault occurs, the allocator makes a copy of the original page and sets the protection level of the faulted page to read-write. When new input arrives, the allocator invalidates and discards some speculative sequences which do not match the new input. For example in Figure 2.10a, if no event activation occurs at t_3, then the sequences corresponding to $f_8'^1$ and $f_8'^2$ are invalid. State changes of the other speculative sequences up until t_3 are committed. In order to correctly roll back a speculation, the allocator copies back the original content of the dirty pages using the copies that it created. The allocator also tracks any pages created as a result of `new` object allocations since the last checkpoint. Any such pages are discarded. During speculation, the allocator also defers page deallocation resulting from object `delete` until commit because deleted objects may need to be restored if the speculation is later invalidated.

For object-level checkpointing, I track lifetimes of objects rather than pages for any object the developer has marked for tracking. To rollback a speculation, I delete any object that did not exist at the checkpoint, and restore any objects that were deleted during speculation. Moreover, I take advantage of *inverse functions* when available to quickly undo object state changes. Many SSOs in games expose custom inverse functions that undo the actions of previous state change. For example, for geometry objects that speculatively execute a $move(\Delta x)$ function, a suitable inverse is $move(-\Delta x)$. Identifying and tracking invertible functions is a trade-off in developer effort; I found the savings in checkpoint and rollback time to be very significant in certain cases (Section 2.9).

2.6 Bandwidth Compression

Navigation and impulse speculation generate additional frames to transmit from server to client. As an example, consider impulse speculation which for RTT of 256ms transmits four speculative frames for four possible worlds. Nominally, this bandwidth overhead is four times that of transmitting a single frame.

I can achieve a large reduction in bandwidth by observing that frames from different speculations share significant spatial and temporal coherence. Using Figure 2.10a as an example, $f_8'^1$ and $f_8'^2$ are likely to look very similar, with the only difference being two frames' worth of a weapon discharge animation in $f_8'^1$. Corresponding screenshots Figure 2.10b–2.10e show that the surrounding environment is largely unchanged, and therefore the spatial coherence is often high. In addition, when Outatime speculates for the next four frames, $f_9'^1$-$f_9'^1$, $f_9'^1$ is likely to look similar not only to $f_8'^1$, but also to $f_8'^2$, and therefore the temporal coherence is also often high. Similarly, navigation speculation's clipped cube map faces often exhibit both temporal and spatial coherence.

Outatime takes advantage of temporal and spatial coherence to reduce bandwidth by *joint encoding* of speculative frames. Encoding is the server-side process of compressing raw RGB frames into a compact bitstream which are then transmitted to the client where they are decoded and displayed. A key step of standard codecs such as H.264 is to divide each frame into macroblocks (e.g., 64×64 bit). A search process then identifies macroblocks that are equivalent (in some lossy domain) both intra-frame and inter-frame. In Outatime, I perform joint encoding by extending the search process to be inter-speculation; macroblocks across streams of different speculations are compared for equivalence. When an equivalency is found, I need only transmit the data for the first macroblock, and use pointers to it for the other macroblocks.

The addition of inter-speculation search does not change the client's decoding complexity but does introduce more encoding complexity on the server. Fortunately, modern GPUs are equipped with very fast hardware accelerated encoders [79, 54]. I have reprogrammed these otherwise idle hardware accelerated capabilities for speculation's joint encoding.

2.7 Multiplayer

Thus far, I have described Outatime from the perspective of a single user. Outatime works in a straightforward manner for multiplayer as well, though it is useful to clarify some nuances. As a matter of background, I briefly review distributed consistency in multiplayer systems. The standard architecture of a multiplayer gaming system is composed of traditional thick clients at the end hosts and a game *state coordination server* which reconciles distributed state updates to produce an eventually consistent view of events. For responsiveness, each client may perform local dead reckoning [34, 14]. As an example, player one locally computes the position of player two based off of last reported trajectory. If player one should fire at player two who deviates from the dead-reckoned path, whether a hit is actually scored depends on the coordination server's reconciliation choice. Reconciliation can be crude and disconcerting when local dead-reckoned results are overridden; users perceive *glitches* such as: 1) an opponent's avatar appears to teleport if the opponent does not follow the dead-reckoned path, 2) a player in a firefight fires first yet still suffers a fatality, 3) "sponging" occurs — a phenomenon whereby a player sees an opponent soak up lots of damage without getting hurt [4].

With multiplayer, Outatime applies the architecture of Figure 2.2 to clients without altering the coordination server: end hosts run thin clients and servers run end hosts' corresponding Outatime server processes. The coordination server — which need not be co-located with the Outatime server processes — runs as in standard multiplayer. Outatime's multiplayer consistency is equivalent to standard multiplayer's because dead-reckoning is still used for opponents' positions; glitches can occur, but they are no more or less frequent than in standard multiplayer. As future work, I are interested in extending Outatime's speculative approach in conjunction with AI-led state approximations [13] to better remedy glitches that appear generally in any multiplayer game.

2.8 Implementation

To prototype Outatime, I modified Doom 3 (originally 366,000 lines of code) and Fable 3 (originally 959,000 lines of code). Doom 3 was released in 2004 and open sourced in 2011. Fable 3 was released in 2011. While both games are several years old, I note that the core gameplay of first person shooters and role playing games upon which Outatime relies has not fundamentally changed in newer games. The following section's discussion is with respect to Doom 3. The experience with Fable 3 was similar.

I have made the following key modifications to Doom 3. To enable deterministic speculation, I made changes according to [25] such as de-randomizing the random number generator, enforcing deterministic thread scheduling, and replacing timer interrupts with non-time-based function callbacks. To support impulse speculation, I spawn up to four Doom 3 *slaves*, each of which is a modified instance of the original game. Each slave accepts the following commands: `advance` consumes an input sequence and simulates game logic accordingly; `render` produces a frame corresponding to the current simulation state; `undo` discards any uncommitted state; `commit` makes any input applied thus far permanent. Each slave receives instructions from *master* process regarding the speculation (i.e., input sequence) it should be executing, and returns framebuffers as encoded bitstream packets to the master using shared memory. To support navigation speculation, I add an additional slave command: `rendercube`, which produces the cubemap and depth maps necessary for IBR. The number of slaves spawned depends on the network latency. When RTT exceeds 128 ms, four slaves can cover four speculative state branches. Otherwise, three slaves suffice. The client is a simple thin client with the ability to perform IBR.

I implemented bandwidth compression with hardware-accelerated video encode and decode. The server-side joint video encode pipeline and client-side decode pipeline used the Nvidia NVENC hardware encoder and Nvidia Video Processor decoder, respectively [79]. The encode pipeline consists of raw frame capture, color space conversion and H.264 bitstream encoding. The decode pipeline consists of H.264 bitstream decoding and color space conversion to raw frames. I implemented the client-side IBR function as an OpenGL GLSL shader which consumes decoded raw frames and produces a compensated final frame for display.

Lastly, I also examined the source code for Unreal Engine [34], one of several widely used commercial game engines upon which many games are built, and verified that the modifications described above are general and feasible there as well. Specifically, support exists for rendering multiple views, the key primitive of `rendercube` [33]. Other key operations such as checkpoint, rollback and determinism can be implemented at the level of the C++ library linker. Therefore, I suggest speculative execution is broadly applicable across commercial titles, and can be systematized with additional work.

2.9 Evaluation

I use both user studies and performance benchmarking to characterize the cost and benefits of Outatime. User studies are useful to assess perceived responsiveness and visual quality degradation, and to learn how macro-level system behavior impacts gameplay. Primary tests are on Doom 3 because twitch-based gaming is very sensitive to latency. I confirm the results with limited secondary tests on Fable 3. A summary of my findings are as follows.

- Based on subjective assessment, players — including seasoned gamers — rate Outatime playable with only minor impairment noticeable up to 128ms.
- Users demonstrate very little drop off of in-game skills with Outatime as compared to a standard cloud gaming system.
- Real-time checkpointing and rollback service is a key enabler for speculation, taking less than 1ms for checkpointing app state.
- Speculation imposes increased demands on resource. At 128ms, bandwidth consumption is 1.97× higher than standard cloud gaming. However, frame rates are still satisfactorily fast at 52fps at 95th percentile.

2.9.1 Experimental Setup

I tested Outatime against the following baselines. *Standard Thick Client* consists of out-of-the-box Doom 3, which is a traditional client-only application. *Standard Thin Client* emulates the traditional cloud gaming architecture shown in Figure 2.1a, where Doom 3 is executed on a server without speculation, and the player submits input and views output frames on a client. The server consists of an HP z420 machine with quad core Intel i7, 16GB memory, and an Nvidia GTX 680 GPU w/4GB memory. For Thin Client and Outatime, I emulated a network with a defined RTT. The emulation consisted of delaying input processing and output frames to and from the server and client by a fixed RTT. The client process was hosted on the same machine as the server in order to finely control network RTT. I also used the same machine to run the Thick Client. User input was issued via mouse and keyboard. I configured Doom 3 for a 1024 × 1024 output resolution. I conducted formal user studies with sixty-four participants overall over three separate study sessions. Two studies were for Doom 3 and consisted of twenty-three participants (Study A) and eighteen participants (Study B). The third was for Fable 3 (Study C) and is described in more detail later in this section. Both Study A and Study B consisted of coworkers and colleagues who were recruited based on their voluntary response to a call for participation in a gaming study. The self-reported age range was 24–42 (39 male, 2 female). The main difference between the studies was that Study B's participants were drawn primarily from a local gaming interest group and were compensated with a $5 gift card, whereas Study A's participants were not. Prior to engagement, all participants were provided an overview of the study, and consented to participate in accordance with institutional ethics and privacy policies. They also made a self-assessment regarding their own video game skill at three granularities: 1) overall video game experience, 2) experience with the first person shooter genre, and 3) experience with Doom 3 specifically. While all Study A's participants reported either Beginner (score=2) or No Experience (score=1) for Doom 3, Study B's participants self-reported as being "gamers" much more frequently, with 72% having previously played Doom 3. For both Study A and B, participants first played the unmodified game to gain familiarity. They then played the same map at various network latency settings with and without Outatime enabled. Participants were blind as to the network latency and whether Outatime was enabled.

2.9.2 User Perception of Gameplay

The user study centered around three criteria.

- *Mean Opinion Score (MOS):* Participants assign a subjective 1–5 score on their experience where 5 indicates no difference from reference, 4 indicates minor differences, 3 indicates acceptable differences, 2 indicates annoying differences and 1 indicates unplayable. MOS is a standard metric in the evaluation of video and audio communication services.

- *Skill Impact:* I use the decrease in players' in-game health as a proxy for the skill degradation resulting from higher latency.

- *Task Completion Time:* Participants are asked to finish an in-game task in the shortest possible time under varying latency conditions.

Each participant first played a reference level on the Thick Client system, during which time they had an opportunity to familiarize themselves with game controls, as well as experience best-case responsiveness and visual quality. Next, they re-played the level three to ten times with either Outatime or Thin Client and an RTT selected randomly from $\{0ms, 64ms, 128ms, 256ms, 384ms\}$. Among the multiple replays, they also played once on Thick Client as a control. Participants and the experimenter were blind to the system configuration during re-plays. Some of the participants repeated the entire process for a second level. I configured the level so that participants only had access to the fastest firing weapon so that any degradations in responsiveness would be more readily apparent.

After each re-play, participants were asked to rank their experience relative to the reference on an MOS scale according to three questions: (1) How was your overall user experience? (2) How was the responsiveness of the controls? (3) How was the graphical visual quality? I also solicited free-form comments and recorded in-game vocal exclamations which turned out to be illuminating. Lastly, I recorded general player statistics during play, such as remaining player health and time to finish the level.

2.9.2.1 Mean Opinion Score

Figure 2.11 summarizes overall MOS across participants from Study A and Study B when playing on Outatime, Thin Client and Thick Client at various RTTs. Thick client is not MOS$=5$ due to a placebo effect. Thin Client MOS follows a sharp downward trend, indicating that the game becomes increasingly frustrating to play as early as 128ms. Free form participant comments from those self-identified as experts strongly reinforced this assessment.

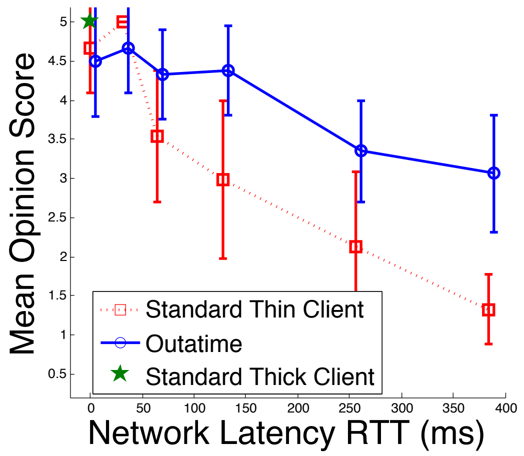


Figure 2.11: Impact of Latency on User Experience

- Thin Client @ 64ms: *"OK, can play. Not acceptable for expert."*
- Thin Client @ 128ms: *"Felt slow. Needed to guess actions to play."*
- Thin Client @ 128ms: *"Controls were extremely delayed."*

For Outatime, the MOS stays relatively high with scores between 4 to 4.5 up through 128ms. Comments from those self-assessed as experts are shown below.

- Outatime @ 128ms: *"Controls were fluid, animations worked as expected, no rendering issues on movement and no transition issues with textures."*
- Outatime @ 128ms: *"Shooting was a bit glitchy, but you don't notice too much. Controls were about the same, good responsiveness and all."*
- Outatime @ 128ms: *"Frame rate comfortable; liked the smoothness."*

Participants with high prior experience tended to assign lower MOS scores to both Thin and Outatime, especially at high latencies. This was evident with participants in Study B who ranked latencies at 256ms merely acceptable or annoying. However, Study B participants still ranked Outatime well at 128ms with a score of 4.4. This

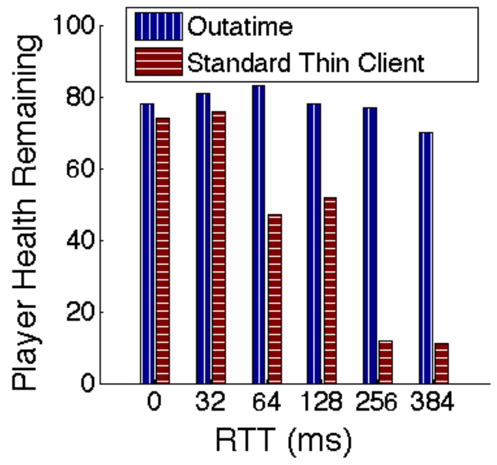


Figure 2.12: Remaining Health

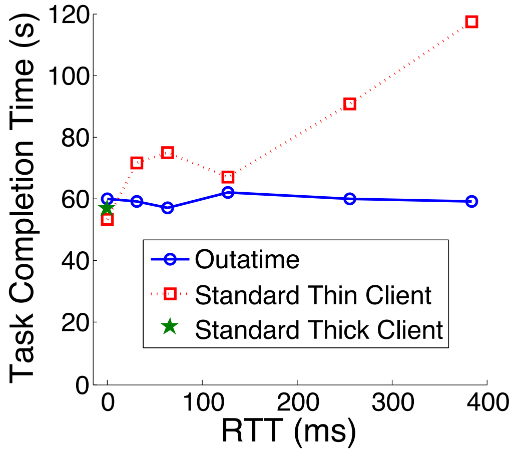


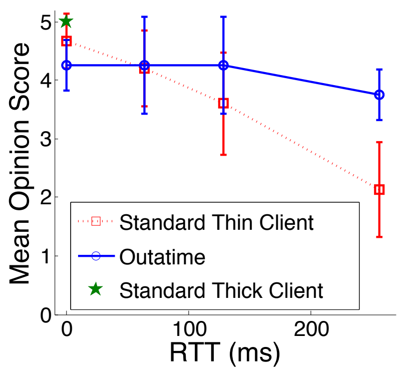
Figure 2.13: Task Completion Time

compares favorably to their corresponding Thin Client at 128ms score of 3.3. On the other hand, even at 256ms, Study A users ranked Outatime as still acceptable at 4.5. This is likely the result of Study A users' less frenetic gameplay, which results in fewer mispredictions. Therefore, I suggest that Outatime is appropriate for novice players up to 256ms, and more advanced players up to 128ms.

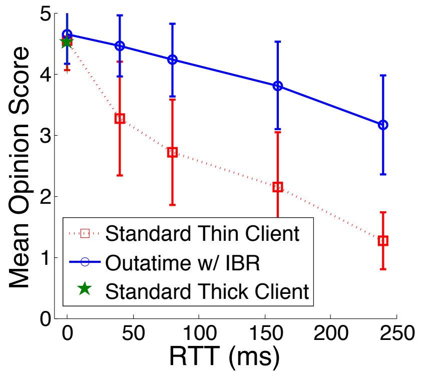
Responsiveness MOS ratings were overall very high (above 4 for all RTTs) for Outatime indicating that Outatime does well at masking latency. Visual quality MOS ratings for Outatime followed the same trends as Overall MOS ratings. The results are elided for space.

2.9.2.2 Skill Impact

I found that longer latencies also hurt performance on in-game skills such as avoiding enemy attacks. For Study A, I instructed participants to eliminate all enemies in a level while preserving as much health as possible. Figure 2.12 shows the participants' remaining health after finishing the level. Interestingly, even though participants playing on Thin Client reported only modest degradation in MOS at 64ms, participant health dropped off sharply from over 70/100 to under 50/100, sug-



(a) Impulse Speculation



(b) Navigation Speculation

Figure 2.14: Fable 3 MOS

gesting that in-game skills were impaired. Outatime exhibited no significant drop off for RTT$\leq$ 256ms.

In the free form comments, several participants mentioned that they consciously changed their style of play to cope with higher Thin Client latencies. For example, they remained in defensive positions more often, and did not explore as aggressively as they would have otherwise. Outatime elicited no such comments.

2.9.2.3 Task Completion Time

Lastly, I measured participants' level completion time. Participants were instructed to eliminate all enemies from a level as quickly as possible. Figure 2.13 shows that RTT $\geq$ 256ms lowered Thin Client completion times, but had little impact on Outatime completion times.

2.9.3 Fable 3 Verification

I setup Fable 3 for similar testing to Doom 3. I recruited twenty-three additional subjects (age 20–34, 4 females, 19 males) who were unfamiliar with the Doom 3 experiments. In this study, I separated the testing of Impule and Navigation speculation in order to isolate their component effects. Figure 2.14a and Figure 2.14b show that the MOS impact of impulse speculation and navigation speculation for Fable 3. Impulse speculation tends to hold up better than Navigation speculation to longer RTTs. This supports my anecdotal observation that visual artifacts from IBR can be noticeable especially over longer time horizons, whereas Outatime's parallel timeline speculation works well to cover all possible impulse events.

2.9.4 System Performance and Overhead

I report a variety of client, server and bandwidth metrics. During server testing, I use a trace-driven client. Similarly, I use a trace-driven server during client tests.

2.9.4.1 Client Performance

Figure 2.15 shows that Outatime and Thick Client both achieve the target frame time of 32ms, which is directly linked to players' perceptions of low latency. The bulk of Outatime's time is spent on decoding, which is dependent solely upon the system's hardware decoder and is not the focus of optimization in this paper. Non-decoding time, primarily IBR, accounts for less than 2ms, even when run on a 2010-year notebook's Nvidia GT 320M, which is 21× less powerful than the server's GTX 680 according to PassMark GPU benchmarks. In contrast, Thin Client's frame time is clearly vulnerable to RTT.

I also tested a configuration that is more representative of a modern high-end game to assess whether it is possible for Outatime to in fact run faster than a Thick Client implementation. Specifically, I modified Doom 3 to run the *Perfected Doom* mod, an enhancement to Doom 3 that increases graphics quality significantly, and more closely matches a modern game.[4]. As shown by Very Thick Client in Figure 2.15, this causes huge spikes in frame time up to 76ms at the 95th percentile, which is unplayable. On the other hand, cloud gaming client performance such as in Outatime is not tied

[4]`http://www.moddb.com/mods/perfected-doom-3-version-500`

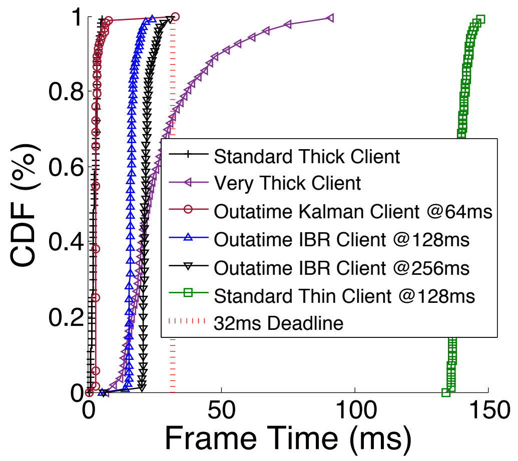


Figure 2.15: Client Frame Time

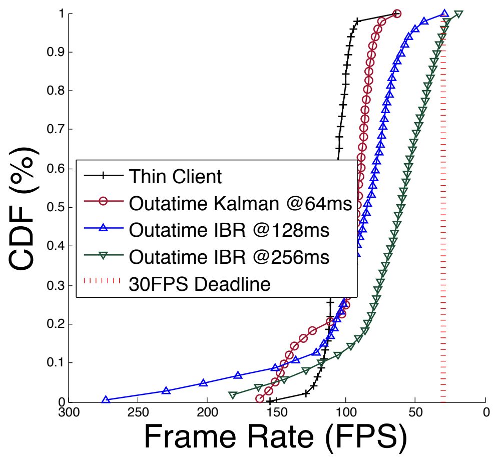


Figure 2.16: Frame Throughput Measured at Server

	Checkpoint Restore	App Logic	Render	Encode	Total (ms)
Thin Client	N/A	0.5 ± 0.1	1.6 ± 0.3	7.3 ± 0.6	9.4 ± 1.0
Outatime					
Kalman@64ms	1.9 ± 0.6	4.6 ± 1.9	1.9 ± 0.3	6.9 ± 1.8	15.2 ± 4.6
IBR @128ms	2.2 ± 0.8	6.7 ± 4.0	2.9 ± 2.1	20.7 ± 7.4	32.4 ± 14.3
IBR @256ms	2.3 ± 1.1	8.8 ± 5.3	3.9 ± 3.2	32.5 ± 12.2	47.5 ± 21.9

Table 2.1: Server Processing Time Per-Frame ± StdDev. Note that Outatime's server processing time is masked by speculation whereas Thin Client's is not.

to the game's complexity. Therefore, the Outatime client can actually outperform the Thick Client in the case of graphically- or computationally-demanding modern games.

2.9.4.2 Server Throughput and Processing Time

I next quantify the server load in two ways. The first is frame rate where I target at least 30fps throughput. The second is the server's per frame processing time. This is the time it takes for input received at the server to be converted into an output frame. Importantly, processing time is not the inverse of throughput because the server implementation is highly parallelized.

Method	Time (ms)
Built-in Save Game	333000
Baseline Object-only	45.55
Outatime's Hybrid	0.076

Table 2.2: Fast Hybrid Checkpoint vs. Alternatives

Figure 2.16 shows the server's frame rate. Outatime is able to operate at above 30fps for every supported latency. In the case of 128ms, the frame rate is 52fps or better 95th percent of the time. Table 2.1 shows the server's processing time. Outatime with Kalman takes longer than Thin Client due to impulse speculation, checkpoint, restore and rollback. Outatime with IBR adds cube and depth map rendering and frame transfer. Processing time is higher at 256ms than at 128ms due to the need to run extra slaves to service more speculative branches. Note that for Outatime, processing time has no bearing on the client's frame time because of Outatime's use of speculation. However, for Thin Client, server processing time (9ms in this case) further increases overall frame time.

2.9.4.3 Checkpoint and Rollback

I perform a comparative benchmark of the fast checkpoint/rollback implementation to two other possible implementations in Table 2.2. The setup is a standalone stress test on Doom 3 where checkpoint speed is isolated from other system effects. The implementation using hybrid object- and page-level checkpointing is approximately 7000× faster than Doom's built-in state-saving Load/Save game feature, and 300× faster than an object-only checkpointing implementation. A similar advantage would exist against page-only checkpointing as well. In absolute terms, tens of milliseconds is simply too costly for continuous real-time checkpointing, whereas sub-millisecond speed makes speculation feasible.

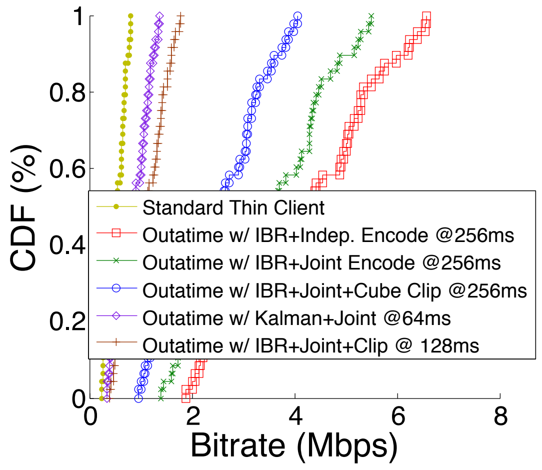


Figure 2.17: Bandwidth Overhead

2.9.4.4 Bitrate

While the prototype performs cube clipping directly on rendered frames and uses a hardware accelerated codec pipeline for efficiency, I conducted compression testing offline using `ffmpeg` and `libx264`, two publicly available codec libraries, for easier repeatability. Figure 2.17 shows the bitrate of Thin Client and various Outatime configurations. The baseline Thin Client median bitrate is 0.53Mbps. When running Outatime with IBR Navigation Speculation and Impulse Speculation with RTT$=$ 256ms, transmission of all speculative frames (cube map faces, depth map faces and speculative impulse frames) with independent encoding consumes a median of 4.01Mbps. After joint encoding, transmission drops to 3.33Mbps. Outatime's use of clipped cube map with joint encoding consumes 2.41Mbps, which is 4.54$\times$ the bitrate of Thin Client and a 40% reduction from independent. Finally, when RTT$\leq$ 128ms, joint encoding and clipping consumes only 1.04Mbps, which is only 1.97$\times$ more than Thin Client. The savings are due to lower prediction error over a shorter time horizon (Figure 2.8) and transmitting half as many speculative branch frames. When running Outatime with Kalman-based Navigation Speculation and Impulse, the bitrate is further lowered to 0.8Mbps, or 1.51$\times$ Thin Client.

2.10 Discussion

Outatime is currently unable to cope with some types of game events. For example, teleportation via wormholes do not map well to IBR's inherent assumption of movement in Euclidian space. One possibility is to extend impulse-like parallel speculation even for navigation events in limited cases. Another interesting avenue of investigation is harnessing additional in-game predictive signals to improve on such speculation. Example signals include player status and visible scene object *e.g.,* presence of wormholes. In addition, applying more powerful predictive time-series models altogether might yield significant gains, especially over longer RTTs. Also, large numbers of simultaneous impulse events over short time horizons limit Outatime's ability to meet high frame rate performance targets. Certain genres of games (e.g., fighting, real-time strategy) exhibit this vulnerability more so than others, so additional state space approximation mechanisms may be needed in such cases [96]. Lastly, support for audio is an area of future work.

Another interesting direction is to explore optimization of parallel speculations. Currently, each speculative branch incurs the full cost of a normal execution. Instead, it may be possible to perform only the "diff" operations between branches (such as rendering only differing objects and inexpensively recompositing the scene at the client), thereby saving significant compute, rendering and bandwidth costs.

I have not focused on client power consumption in this work, but it is potentially a very ripe area for savings. This is because Outatime's thin client approach essentially converts the client's work from an arbitrary app-dependent rendering cost to a fixed IBR cost (see Figure 2.15) since IBR cost is only dependent upon the client screen resolution. The potential for implementing IBR as a highly optimized silicon accelerator is very intriguing.

Despite of these limitations, Outatime illustrates the promising result on improving usability of cloud-based gaming application even in the presence of high latency wireless network.

CHAPTER III

Verifying User Interface Properties for Vehicular Applications

Next, I switch my focus from the mobile network to attention-limited mobile environment. In this chapter, I focus on how to improve the usability of mobile applications in an attention-limited environment (i.e., vehicular environment). Some mobile applications are often used in environments in which the user's attention must be focused on driving. For instance, drivers employ GPS route planners, music services, gas station locators, parking-spot finders, and similar tools. While well-designed applications that minimize driver distraction can improve the driving experience, poorly designed applications can too easily divert attention from the primary task of operating the vehicle. Although vehicular applications are currently the primary reason why mobile interfaces are currently designed with user distraction in mind, emerging technologies such as augmented reality are likely to increase the prevalence of such situations.

Unfortunately, as I show in a study of Android applications, many tools intended for vehicular usage today do not follow accepted industry guidelines. I argue that this problem does not arise from a lack of usability research; in fact, there exists a large body of detailed guidelines and standards for user interaction in vehicular environments [29, 45, 77, 97, 104]. Rather, the issue is that most mobile application developers simply do not have enough experience creating interfaces for automotive environments. They are not aware of the best practices in vehicular interface design, and they are not able to anticipate how drivers will interact with their applications in vehicular settings. This information gap results in applications that inappropriately demand too much attention from drivers.

Currently, there are several models for running mobile applications in vehicles. Some applications run on an in-vehicle *human-machine interface* (HMI) such as MyFord Touch or the Cadillac Cue system. Access to such platforms is closely controlled by vehicle manufacturers; any applications developed by third parties undergo strict testing in cooperation with manufacturers before they are deployed. Usability experts ensure that applications meet guidelines for in-vehicle use on a manufacturer's HMI platform. The advantage of this approach is that deployed applications meet a very high usability standard. The disadvantage is cost and speed of deployment; application testing can take many iterations between developers and usability experts. Developers must wait for feedback after each modification to the application interface, and each application evaluation requires a substantial amount of costly expert time.

A more recent class of in-vehicle applications run on smartphones. These applications are deployed through application marketplaces without the involvement of vehicular manufacturers. The advantage of the smartphone model is speed of deployment; mobile developers can often release a new application months after conceiving an initial idea. The disadvantages of this approach are two-fold. First, the input and output capabilities of a smartphone platform are not ideal for the vehicle; in contrast, vehicle HMIs have dashboard touchscreens, steering wheel controls, and other forms of interaction specifically designed for use in a vehicle. Second, as previously mentioned, mobile application developers are unaware of best practices for vehicular applications and often make poor design choices that lead to too much distraction when the application is used.

Recently, a hybrid model of application deployment has emerged in which an application running on a smartphone seamlessly interacts with in-vehicle HMIs by, for example, displaying the application graphical user interface (GUI) on the dashboard touchscreen and by enabling input via steering wheel controls. Ford's AppLink platform and the MirrorLink standard are two examples of this hybrid approach. While such platforms are a promising method for addressing the input and output limitations of smartphones, they provide little help to developers struggling to design applications appropriate for vehicular environments. Even if expert guidance were available to such developers, it is infeasible to have completely-manual testing scale to check the total number of vehicular applications in a marketplace such as iTunes or the Android store.

In this thesis, I mitigate this dilemma with automated model checking of user interfaces. I have developed a tool, called AMC (Android Model Checker) that automatically explores the set of GUI screens displayed by an Android application and verifies that desired user interface properties hold over all such screens. AMC does not require application source code, nor does it require any application-specific knowledge. Instead, AMC develops a model of the application's GUI by executing the application. AMC regards each unique screen as a state, and it considers UI events (e.g., button presses) as transitions between those states. It explores the application state space (the GUI), until it has exhausted all reachable state transitions. It verifies properties such as minimal use of animation, appropriate color contrast, button closeness on the touchscreen, and task complexity as measured by the number of user actions required to complete an action. AMC outputs a list of violations of best practice guidelines for these properties along with snapshots that detail the specific violations.

The goal of AMC is not to completely eliminate the need for testing by human experts. Some issues defy automated judgment and are difficult to quantify. For instance, most experts would consider a moving graph of instantaneous gas mileage to be acceptable for vehicular usage, but would clearly consider a game with animation to be unacceptable. Automatic differentiation of the two animations can be difficult.

Instead, the goal of AMC is to substantially reduce the amount of work that a human expert must perform to help developers verify an application. AMC can check “easy” properties automatically, eliminating such tasks entirely from the expert’s purview. For more challenging properties, AMC can prune large numbers of states and identify a small set of questionable screens that an expert must check manually. This helps the expert focus her time on the tasks that most require human judgment.

AMC also helps application developers by providing them with early feedback about usability violations. A developer can run AMC, generate a list of best practice violations, and fix those bugs without submitting the application for expert testing. After the application passes all AMC automated checks, expert checking can reveal any remaining problems.

To evaluate the tool, I studied 13 popular applications in the Android marketplace: 8 of these applications are designed for vehicular usage and 5 are not. Unsurprisingly, I found that applications that are not specifically intended for vehicular usage often do not follow accepted industry guidelines for vehicular interface design. However, I were surprised to find that *the Android applications intended for vehicular usage violated best practice guidelines at approximately the same rate as non-vehicular applications*. These results highlight the critical need for tools such as AMC that can help developers understand why their user interfaces may be inappropriate for use by the driver of a vehicle.

I evaluated AMC by comparing its results to those manually generated by an industry expert checking the same applications. AMC generated a definitive assessment for 85% of the application/property combinations that I studied. Compared to the expert assessment of these combinations, AMC had no false positives and a false negative rate of less than 2%. AMC deferred the remaining 15% of application/property combinations for further assessment by a human expert, but it reduced the number of screens that an expert must check in such cases by 95%. The median time for AMC to completely explore an application GUI is approximately 73 minutes; the most complex application that I studied (Twitter) required approximately 10 hours.

3.1 Overview

AMC automatically explores the GUI of an Android application. It checks the interface for violations of a set of pre-specified design properties. AMC uses a model-checking technique that attempts to exhaustively explore all screens that can possibly be displayed by the application. For each property, it may report that the property is upheld by the application (no violations were found), or it may output a list of violating screens annotated with screen snapshots that demonstrate each violation. Occasionally, AMC may be unable to automatically determine whether a property is upheld on a particular screen; in such cases, it reports a possible violation with an annotated snapshot.

While the properties that AMC checks are pre-configured, AMC contains no application-specific logic. Testing is fully automatic once started, so a user can leave AMC running and come back later to collect results. For applications with a cloud component, AMC requires that the testing computer have a working Internet connection; AMC also requires that any needed authentication be completed before testing starts. These requirements are minimal so as to make it trivial to apply AMC to testing a new application.

I envision two classes of users. Applications developers will use AMC to get early feedback on the distraction properties of their application GUI. A developer can run AMC in his own testing environment and get valuable insights without having to wait for feedback from a human expert.

I also envision that experts who control access to a vehicular application marketplace will use AMC to magnify their productivity. Many properties such as a maximum number of user actions per task and a minimum button size are tedious to check manually on each application screen, yet these properties are easy for a computer to verify automatically. I expect that an expert will simply rely on AMC to check these types of properties. Other properties, such as the acceptability of animation, require human judgment. AMC can classify some simple cases; for the remaining cases, it enumerates the specific screens that the expert should check further. Thus, even for properties that are hard to verify automatically, AMC can still substantially reduce the testing time required of a human expert.

3.2 Design and implementation

AMC verifies user interface properties with a model checking approach similar to that used by CMC [76] in which it directly checks the implementation of an application. AMC starts with no information about the user interfaces states and the transitions among those states; instead, it discovers this information on the fly by triggering user actions and observing their effect on the application GUI.

Most prior model checkers operate by taking checkpoints of application states to allow the checker to deterministically revert to any past state by restoring a checkpoint. I initially planned to use this approach in AMC, but I quickly realized that checking mobile applications presents a substantial complication: the client typically only contains a portion of the application state, with the remainder contained in a cloud component. Restoring only the mobile component to a prior state is infeasible because the mobile and cloud components become out-of-sync. It is rare to see a mobile application in which the interaction between client and server is truly stateless (for instance, encrypted connections have inherent state, caching introduces state dependencies, and even simple identifiers often have constraints such as increasing monotonically).

AMC could potentially take a coordinated checkpoint on the mobile client and the cloud server. This is infeasible for third-party testers who do not have access to the cloud component. Even developers may lack the ability to checkpoint cloud state if the cloud component runs in an environment hosted by another organization.

For these reasons, AMC instead explores application state without the use of checkpoints. It uses the application GUI to return to desired states. This requires that AMC use its model of states and transitions among those states for navigation. It also requires that AMC discover when a new transition leads to a state that it has seen previously. This section discuss how AMC handles these and other challenges.

3.2.1 Modeling the user interface

AMC represents the graphical user interface of an Android application as a finite-state model. Intuitively, each distinct screen of the application is a state, and the user actions that transition between those screens are state transitions. AMC's model is therefore a directed graph in which the distinct screens are vertexes and the user actions are edges. In practice, the notion of what represents a "distinct" screen is open to interpretation, and AMC uses state equivalence heuristics, described in the next section, to determine whether two screens with different content and structure should be classified as the same state or as different states.

AMC constructs its state model on the fly. It begins with a single state that represents the initial screen displayed by the application. AMC adds a transition for each user action that may be performed on that screen such as a GUI button. Each of these transitions is initially marked as being unexplored.

AMC's basic state exploration algorithm works as follows. If there are no unexplored transitions left in the state model, AMC has fully explored the application GUI, and so it terminates. Otherwise, AMC chooses an unexplored transition. It orders the unexplored transitions by the number of actions required to reach that transition from the current application state. For instance, unexplored transitions on the screen currently being displayed are given top preference, transitions on any screen that can be reached using one user action are given the next highest preference, and so on. AMC determines the number of user actions to reach a particular state by executing a breadth-first search on the graphical representation of the state model starting at the current application state and terminating when the first unexplored transition is encountered.

After choosing an unexplored transition, AMC generates user events to navigate to the screen containing that transition (unless the transition is located on the current screen, in which case no navigation is required). AMC uses the Android monkey package [113] to send user actions such as touchscreen events and button presses to the Android screen. AMC determines the screen coordinates for such events by parsing the DOM tree, which describes all currently displayed widgets and their relative locations.

AMC next performs the user event necessary to effect the transition and observes the effect on the application GUI. Some actions do not cause the application to display a new screen; AMC models such actions as explored transitions that start and terminate in the same state. Some actions transition to a state that has already been observed by AMC (I defer the discussion of how AMC determines state equivalence to the next subsection). AMC simply adds the explored transition as a vertex from the old state to the new one. The remaining transitions generate a state previously unseen by AMC.

When AMC discovers a new state, it adds the state to its model, marks the transition it just took as explored, and sets the destination of the transition to the new state. It then takes a snapshot of the current screen using the screen capture feature of the Dalvik Debug Monitor Server, which is part of the Android SDK library. Snapshots are saved and may later be used to explain observed violations to the user.

Next, AMC verifies user interface properties, as described in Section 3.2.5. Some properties, such as button size, pertain to individual states. For these properties, AMC highlights the violating region on the screen snapshot for that state in its report. Other properties, such as the number of user actions required to perform a task, are a property of multiple states and transitions. AMC generates a series of snapshots that illustrates the violation in such cases. I have found annotated snapshots to be an effective mechanism for quickly communicating violations to the user of my tool.

3.2.2 Determining state equivalence

AMC's model checking algorithm requires a good metric for determining state equivalence. To understand why this is crucial, consider a straw man equivalence function in which two GUI screens are considered to be the same state if and only if they have the exact same content.

AMC uses the Hierarchy Viewer tool [48] to extract the DOM information, expressed in XML format, for the currently displayed Android screen. The DOM tree contains all of the GUI widgets on a given screen along with any content within those widgets. For instance, the DOM tree will have a node for a list box element, as well as a node for each list item that describes the item contents. AMC generates a unique identifier for each screen, called the *content hash* by traversing the DOM tree in depth-first order and computing a hash over all nodes. Thus, AMC could potentially define two screens to be the same state if they have matching content hashes and declare them to be different states if their content hashes do not agree.

However, the proposed state equivalence function works poorly for most applications of reasonable complexity. Even simple actions, such as adding an item to a list or changing displayed text for an object will lead to a new state. For applications with cloud components, simply refreshing the same screen and downloading new content will lead to a new state. As a result, AMC state exploration will never terminate. Even worse, AMC may spend unnecessary time exploring "new" states that are essentially identical to states that it has seen previously. Thus, strict content equivalence is a poor choice for a state equivalence function.

Ideally, I would like to define state equivalence to match a user's intuitive notions of what constitutes the unique screens in an application. For instance, I would prefer that AMC treat the business and sports section screens of a newsreader application as the same state since both simply provide a list of articles that can be accessed. AMC therefore uses a state equivalence function that is based on structure, not content.

When a screen is displayed, AMC computes its *structure hash* by traversing the DOM tree in depth-first order and hashing the tag name, level, and sibling order of each XML node, omitting all other content. AMC uses the structure hash as a unique identifier for each state. Thus, if two screens have the same structure (layout of buttons, lists, text fields, custom views, etc.), but the screens have different content, AMC considers the two screens to be the same state.

From the initial experience using AMC, I learned that the structure hash is still too strict an equivalence check. As defined so far, the structure hash considers two screens to be different if a list or other container has a different number of elements. For instance, a screen that displays 3 gas stations would not be equivalent to another screen that is identical except for displaying 4 gas stations. This problem arises because individual list items are XML nodes and therefore affect the structure hash. Unfortunately, if this problem is not addressed, the AMC algorithm will sometimes fail to terminate (this is the familiar "state explosion" problem in model checking [114]).

On the other hand, I do not want to omit considering such list items entirely. Having one or two items in a list often enables actions that are not accessible if no items are in the list (for instance, one cannot click on a list item if none are present). Further, some lists such as menus do not contain items of homogeneous type; clicking on different items will have different effects.

AMC uses the following heuristic to handle lists and similar containers. If a list has more than one item, AMC checks whether those items are homogeneous. If the list items have registered different callback handlers, then they are considered heterogeneous (since clicking on different items is likely to have different effects). Since AMC does not require application source code, it distinguishes between two callback handlers by using the binary callback address passed to the GUI system.

Some applications multiplex many actions through the same handler. Therefore, if the callback handler is the same, AMC clicks on the first 2 items. If the items transition to different states, then the list items are deemed heterogeneous; otherwise, they are homogeneous. When AMC sees a list with more than 2 homogeneous items, it canonicalizes that state by considering only the first 2 items in the structure hash. This has the effect of making all instances of the same screen with two or more list items be the same state. Screens with different numbers of heterogeneous elements are always considered to be distinct states.

3.2.3 Non-deterministic transitions

From the point of view of AMC, user actions can sometimes appear to be non-deterministic. That is, taking the same action from the same state transitions to different states at different times. There are several reasons why this behavior can arise. In rare cases, the application may be truly non-deterministic. More often, the state transition is deterministic, but the resulting state depends on some aspect of application state that AMC cannot observe because it is unaware of application semantics. For instance, the back button may transition to a different state depending on the path that the user took to get to the current state. In another example, changing a configuration option in a menu may cause an action to transition to a new state. Mistakes in state classification can also lead to seemingly non-deterministic behavior. For instance, if AMC classifies two different screens as belonging to the same state, it is possible that performing the same action on each screen leads to a different state. While such misclassifications are rare, they will inevitably occur in any model-checking system like AMC that uses heuristics to avoid state explosion.

Seemingly non-deterministic behavior complicates the basic search strategy described in Section 3.2.1. When AMC performs user actions to transition to the nearest state with an unexplored transition, it may go astray and arrive at a different state than expected. Therefore, AMC must enhance its basic search algorithm to deal with such behavior.

AMC considers all transitions to be deterministic unless proven otherwise. AMC maintains a second graph of the application state model, called the *exploration graph* — this graph contains only deterministic edges. New edges are added to the exploration graph when AMC first explores the transition. However, if AMC re-traverses an edge in the exploration graph and the transition leads to a state other than the one expected, it labels the transition as non-deterministic and removes the edge from the exploration graph.

AMC calculates the path to the nearest unexplored transition by applying the breadth-first search algorithm described in Section 3.2.1 to the exploration graph. Effectively, this finds the shortest path that traverses only deterministic edges. When the path traversal fails due to a newly-discovered non-deterministic edge, AMC reruns the shortest path algorithm after the edge has been removed.

If too many edges are removed due to non-determinism, AMC may not be able to explore some transitions because it can no longer find a deterministic path to the state that contains the transition. However, I have observed that most seemingly non-deterministic transitions are fully deterministic if the application performs an identical sequence of n user actions (where n is greater than one). As a simple example, the back button often transitions to the state explored prior to the current state. Thus, clicking the back button on a given state appears to be non-deterministic if there exists two or more possible prior states. However, if one considers a 2-step sequence of actions, i.e., transitioning from the prior state to the current state and then clicking the back button, then the transition as a result of that sequence appears completely deterministic.

If after exploring all transitions that can be reached via deterministic transitions, AMC still has an unexplored transition, it uses the following algorithm to try to reach the state containing that transition, which I refer to the goal state. AMC keeps a log that contains the sequence of all states that it has visited along with each transition that it has triggered. It searches through the log to find all occurrences of the goal state. For each occurrence, it observes the past n actions that it took to reach the goal state (n is initially 2). It assumes this sequence is deterministic unless proven otherwise. So, it inserts in the exploration graph an edge from the starting state of the sequence to the goal state. The edge is given a weight of n since it contains n user actions. The edge remains in the graph until the sequence is demonstrated to be non-deterministic. AMC then restarts exploration and tries to reach the goal state via the new paths that it has added. If AMC again fails to reach the goal state, it tries again with $n = 4$ and $n = 8$. After $n = 8$, AMC gives up on exploring this particular transition and reports the state containing the transition as being only "partially explored". If there are multiple unexplored transitions, AMC proceeds until it either has explored or has given up on all such transitions.

3.2.4 Complications in state exploration

The Android platform has a few peculiarities that complicate AMC state exploration. First, Android does not provide a callback to an external observer when the display of a screen completes. Therefore, AMC has no means of knowing when to compute the structure hash. Initially, I used a simple timeout approach in which AMC waited 10 seconds for each screen to stabilize before computing the hash. However, this strategy substantially increased AMC's state exploration time since AMC waited after every transition. I could not reliably set the timeout to a lower value because some slow applications take several seconds to fetch data from a remote server and display results.

AMC currently uses a modified timeout-based approach. After exploring a transition, AMC continuously computes the new screen's structure hash. If the hash matches that of a previously explored state, then AMC considers the screen to have stabilized and waits no longer. With this approach, AMC waits for the full 10 second timeout only once for each new application state, and not once for every transition. This substantially decreases state exploration time because the number of states is much less than the number of transitions.

Many Android application screens have components that allow users to scroll down to view more items than can be displayed at one time. Depending on the specific method used by developers to implement scrolling, items not currently displayed may or may not appear in the DOM tree. For example, elements not currently displayed appear in the DOM Tree when the Android ScrollView widget is used, but not when the ListView widget is used. In the former case, AMC does not need to perform any extra actions. In the latter case, AMC sends events to the screen to cause the application to scroll down until all elements have been displayed. AMC identifies new elements that appear during scrolling by their unique row identifier; it adds these elements to its internal representation of the screen's DOM tree. Once the DOM tree is complete, AMC proceeds as described previously.

Some applications require text input to operate. The two examples I encountered were passwords and destination fields for GPS applications. In such cases, the AMC user must supply the appropriate text to enter and the associated text field's id prior to running AMC. With this information, AMC automatically inputs the supplied text on user's behalf whenever it encounters the associated text field on a screen.

3.2.5 Verifying user interface properties

User interface guidelines for vehicular settings are based on two key principles. First, vehicular application should not give rise to potentially hazardous behavior by the driver or other road users [104]. The driver's primary focus must be on the road and the driving task; the vehicular application's user interface should minimize its distraction level. This principle bounds the amount of time that the driver can spend on each action. For instance, as button size decreases, Fitts' Law [37] predicts that it will take longer for the user to position her finger to touch the button. Thus, overly small buttons require too much attention and distract driver focus from the road.

Additionally, actions should not have time limits since a driver should not be forced to divert his attention from the road at inappropriate moments. Many forms of animation violate this principle, since a user must watch the screen at a particular time to gather needed information.

Another principle is that that application must be consistent and limit state that the driver has to remember [45]. Being consistent includes maintaining the general look, feel, and layout of the application. For example, buttons should be placed in the same portion of the screen, with the same general functionality, so that the user can easily navigate. Limiting state bounds the maximum number of actions that can be required to perform a task.

AMC currently verifies seven properties: number of actions to perform a task, text contrast ratio, word count per screen, button size, button closeness, use of animation, and the amount of scrolling required per screen. AMC presents a modular interface for adding new properties, and it separates the verification code from the state exploration code. This design has made it easy to add new property verification modules as the experience with the tool has grown. I next discuss each property that AMC verifies in more detail.

3.2.5.1 User actions per task

Most vehicular interface guidelines suggest limiting the number of user actions (button presses, touchscreen events, etc.) that a driver must perform in order to complete a single task [29, 77, 97]. Excessive sequences of user actions increase the cognitive load on the driver and divert too much visual attention from the road. Since existing guidelines do not agree on a specific maximum number of actions that should be allowed, I consulted with an expert in Ford Motor Corporation's HMI lab, who suggested that no task should take more than 10 actions to complete.

While the notion of a task is somewhat application-specific, AMC can generate reasonable approximations of tasks using the graphical model created by state exploration. Given a starting state, a task is a series of actions that transitions to some destination state and then returns back to the starting state. For instance, a task could be navigating through menus and sub-menus, setting a configuration option, and then returning via the back button.

For every state in the graph, AMC finds the shortest round-trip path to and returning from every other state. If the length of this path is greater than the limit of 10 actions, AMC reports a violation. It appends annotated snapshots of all screens along the violating path and the transitions that move from one screen to another. This check is performed after state exploration finishes. Detecting violations does not require revisiting states since all necessary information is contained in the graphical state model.

AMC exhaustively considers all tasks rather than attempting to identify the set of tasks that are performed frequently. Such identification would require application-specific knowledge that AMC does not possess. Further, even infrequent tasks that violate guidelines should be avoided.

3.2.5.2 Text contrast ratio

Drivers should be able to discern screen elements with a quick glance. A poor contrast ratio between elements displayed on a screen increases the amount of time and attention required to understand the screen contents. Consequently, a contrast ratio of at least 3:1 between the luminance of the foreground and background is recommended [97]. AMC currently verifies this property for all text elements displayed by an application.

AMC uses the snapshot of each state to calculate contrast ratios for text elements. It parses the DOM tree to find all view objects that have text elements. It then extracts color information and screen locations for all such objects. AMC computes the background and foreground RGB values for each element, and it converts these values into luminance values using a conversion formula proposed by W3C's Ib Content Accessibility Guidelines [20]. If the contrast ratio is below 3:1, AMC reports a violation and highlights the violating element on the state snapshot.

3.2.5.3 Text word count

Excessive text on a screen requires too much driver attention. According to industry-standard user interface guidelines [29], a task should take no longer than 20 seconds to finish. The average adult reads approximately 250 to 300 words per minute [122]. Thus, no screen should ever contain more than 100 words. Using this as a conservative upper bound (it is trivial to set the threshold to lower values), AMC counts the number of words in all text elements and reports a violation if there are more than 100 total words for any state.

Applications frequently embed text in images (for example, by displaying logos or using buttons with bitmap images). When text is embedded in an image, the DOM tree does not provide a description of that text. AMC handles these cases with Optical Character Recognition (OCR). When AMC encounters a leaf in the DOM tree that does not have a text element, it first calculates the screen region for that element. It then uses the Tesseract OCR tool [107] to identify any text within that region. If AMC discovers text, it adds the word count for that text to the total word count for the state.

If the screen has a scrolling component that hides some elements from view, AMC scrolls down to display more elements. This continues until all elements have been displayed or until the word count exceeds the specified limit.

3.2.5.4 Button size

Initiating touchscreen events should not be excessively distracting. If the surface area that initiates an event is too small, the driver must focus her attention on making sure that her finger contacts the screen at exactly the right location; this diverts attention from the road. Consequently, best practice guidelines say that the minimum contact surface area for a control such as a button is $80\,\mathrm{mm}^2$ [97].

AMC verifies this property by first determining the height and width of a pixel on the test platform. In the evaluation, I assume a 9-inch dashboard touchscreen. This is conservative for Android applications since most smartphone screens are substantially smaller. For each clickable item on the screen, the DOM tree reveals the height and width of the item in terms of the number of pixels. By multiplying these values and scaling by the platform-specific conversion factors, AMC calculates the absolute button size. If this is less than $80\,\mathrm{mm}^2$, it reports a violation and highlights the button on the application screen snapshot.

Android allows applications to add their own window element types; these application-specific types are referred to as *custom views*. For instance, many vehicular applications use a custom view to display a map element. Custom views often contain clickable button-like elements. Unlike standard buttons which are shown as elements in the DOM tree, individual clickable elements within an opaque custom view are hidden from AMC. AMC can detect the existence of a click handler function, but it cannot determine the size or location of clickable regions from the DOM tree. It cannot even determine the number of clickable regions in the custom view.

AMC uses empirical testing to learn more about clickable elements in a custom view. It sends click events to a custom view at $< x, y >$ coordinates that are laid out in a grid pattern such that the horizontal and vertical difference between clicks is slightly less than 50% of the minimum button size given a square button. For the 9-inch touchscreen, this works out to a grid with coordinates 20 pixels apart.

AMC then clicks on each grid coordinate. If the click causes the application to transition to a different state, AMC records the state at which it arrived, then returns to the original state to click on more grid coordinates.

AMC considers any set of contiguous grid coordinates that transition to the same state to be part of the same clickable region (button). For simplicity, it currently assumes that such regions are rectangular. Because the grid is regularly spaced, AMC can bound the minimum and maximum dimensions for a region. For instance, a region that is two clicks in width must have width between 21 and 59 pixels. AMC calculates the minimum and maximum button area from this information. If the maximum area is less than $80\,\text{mm}^2$, a violation is reported. If the minimum area is at least $80\,\text{mm}^2$, no violation exists. Otherwise, AMC needs more information to make a determination, since the button area could either be bigger or smaller than the minimum area. It therefore clicks on additional $< x, y >$ coordinates in a binary search pattern to determine the height and/or width of the region. The search terminates once a violation is detected or the button is determined to be at least as large as the guidelines require. In other words, AMC narrows its estimate of the button size until it can precisely determine whether or not the button violates the guideline.

This method of determining region size relies on the custom view not changing between clicks. Once the view changes, button locations may be added, removed, or moved; in other words, AMC can infer no useful information about region size. AMC checks to see if a custom view changes by comparing screen snapshots of the view region. It takes an initial snapshot before sending any click events. Five seconds after each click, it takes a new snapshot of the view region and compares that with the original snapshot. If the snapshots differ by more than 1%, AMC declares the view to have changed. In this case, AMC reports the view as a potential violation that requires further analysis because it cannot make a definitive judgment about the clickable regions contained within the view.

3.2.5.5 Button distance

For reasons similar to those cited for button size, the distance between the center of each pair of buttons should be at least 15 mm [97]. AMC determines the geographical center of each button on the screen, calculates the Euclidean distance in terms of the number of pixels, converts that value to a screen-specific absolute distance value, and reports a violation if the limit is exceeded.

For custom views, AMC uses a similar strategy to the one used to determine button size. It first uses the grid information to determine bounds on the center of each clickable region (again, assuming that such regions are rectangular). If all regions can be determined to be more than the maximum distance from each other, AMC terminates without reporting a violation. If any pair of regions are definitely too close to each other, AMC reports a violation. If the grid has insufficient granularity to determine whether a pair of regions are too close, AMC performs binary search to refine the bounds on the center of each button until it can determine whether or not a violation exists.

3.2.5.6 Animation

Drivers must be free to interact with the application at a time of their convenience. No application should ever demand that an action be performed within a time limit, since performing the action may force the driver to divert attention form the road at an inappropriate instance. A corollary of this guideline is that an application should never display animations, videos, and other highly-interactive visual elements [29].

Unlike the previous guidelines, human judgment is usually required to determine whether a visual artifact is acceptable. For instance, an animated graphical display of fuel efficiency in a hybrid vehicle is typically considered to be acceptable. On the other hand, interrupting a route display in a GPS application to pop-up an unsolicited chat message from a nearby vehicle would almost certainly be considered poor vehicular interface design (note: this is an actual application behavior that I observed!). AMC therefore detects *potential* violations of this property, but defers the actual evaluation to an expert tester. The goal of checking this property is therefore to substantially reduce the number of states that such an expert needs to examine.

When AMC discovers a new state, it pauses for four seconds to detect possible animations. During this interval, it takes several screen snapshots. It breaks each of the screen regions into blocks using a grid pattern and calculates the Sum of Absolute Differences (SAD) on each color channel to determine if a block has changed significantly between snapshots. Any block that has SAD value more than zero on any color channel is considered a potential violation. The changing regions on the screen snapshots are highlighted to illustrate the potential violation.

If an animation only affects a small portion of the screen, it is unlikely to be distracting. For instance, a blinking cursor can be helpful in drawing attention to an important screen region and may help reduce cognitive load.

When AMC detects animation on a screen, it divides the screen into 400 equally-sized blocks. It then compares each of these regions from the two snapshots. If content has changed in less than 10% of the blocks, AMC determines that there is no violation. If more than 10% of the content has changed, AMC reports the state as a potential violation. This rule substantially reduces the number of states that need to be manually examined and has not led to false negatives in any application.

3.2.5.7 Scrolling

Scrolling through many options commands too much attention from a driver. Usability guidelines therefore recommend eliminating scrolling or limiting the number of items in a list [97]. Alternative selection approaches such as voice recognition can be used when options are too numerous to include in a scrolling control.

AMC verifies that no list element contains more than 10 elements — I chose a limit of 10 based on a recommendation from an industry expert. For each list view object in a state's DOM tree, AMC reads the value of the 'mItemCount' field. If this value exceeds the limit, AMC reports a violation.

3.3 Evaluation

I evaluated AMC by comparing its results to those of an industry expert evaluating 12 popular Android applications. I also calculated metrics for AMC such as state coverage and exploration time. Finally, I assessed the need for a tool such as AMC by examining how often current Android applications intended for vehicular settings violate best practice interface design guidelines.

3.3.1 Setup

I executed AMC on a Nexus One phone running Android 2.3.4. I have modified the Android system to enable AMC to extract event handler and text color information.

I assumed the existence of a phone-HMI hybrid interface that replicates the smartphone's display on a 9-inch vehicle touch screen. I further assumed that all user interactions with the application will take place using the vehicular touch screen.

Note that a 9-inch display is larger than that of almost all Android phones. Therefore, buttons and icons will be significantly larger on the touch screen. Any size-related violation I identify assuming the existence of a touch screen would also be a violation on most any Android phone.

Some applications support multiple input modalities for initiating an action, for example, UI events and voice commands. I check the UI in such instances because some set of users will opt for touch screen interaction, and so such interaction should meet best practice guidelines.

I selected 14 applications for the evaluation. These were selected because they were the most popular applications in their respective categories in the Android store as of November 2011. Five applications (Gmail, Google Finance, MyDays, eBay, and Twitter) are not targeted for use in vehicular environments. I expected that such applications would have numerous usability violations because they have not been designed for vehicular usage.

I defined eight of the applications (GasBuddy, Google Maps, Google Navigation, TomTom Navigation, Waze, iRadar, Beat the Traffic, and Best Parking) to be *vehicular applications*. Informally, a vehicular application is one that delivers information that may be particularly relevant to the driver of a vehicle, such as gas prices at nearby stations, recommended routes, police radar locations, and traffic information. Some applications, such as Waze, have specific mechanisms such as voice control for minimizing driver distraction. Although other applications, such as Google Maps, are not solely intended for use by a driver, the experience is that many people use such applications while driving because of the helpful information they provide. I expected that these applications would have considerably less violations than the first set (although my expectation turned out to be wrong).

Finally, I also selected an Android application called MPG that was developed as a demonstration application for the Ford OpenXC platform in conjunction with Ford experts who are well-versed in vehicular user-interface guidelines. This application was created specifically for a vehicle touch screen by a University of Michigan undergraduate student who was not involved in the AMC project.

3.3.2 Comparison with industry expert

I evaluated the quality of AMC's results by comparing them with the results of a manual evaluation by an industry expert. Since the expert had limited testing time available, I took several steps to reduce the manual testing time. First, I compared only 5 of the 7 categories reported by AMC (text word count, button size, button distance, animation, and scrolling). Manually evaluating the number of user actions per task was too time-consuming because the expert would have to reason about all possible paths through the application to evaluate each action. The expert did not evaluate contrast ratio because the measurement requires special software tools that were not available at the time.

Second, the expert evaluated only 12 of the 14 applications, omitting Gmail and Google Maps because they were too complex to evaluate thoroughly given a limited time budget. Finally, rather than ask the expert to exhaustively count all violations of each property in each application, I asked the expert to render only a binary decision: either the application upholds the property or it contains one or more violations.

Prior to manual testing, I informed the expert of the specific properties that I were checking for violations. However, I did not disclose AMC's results to the expert prior to testing. After the expert completed an independent assessment, I compared results for each application. In a few cases, AMC found violations that the expert did not discover (usually due to limited testing time), but which the expert agreed were indeed violations. I used this revised expert assessment as ground truth (i.e., I suspect the expert would have found these given unlimited testing time).

	Text word count		Button size		Button distance		Animation		Scrolling	
Popular Apps	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert
Google Finance	×	×	×	×	×	×	✓	✓	×	×
MyDays	×	×	✓	✓	×	×	✓	✓	×	×
eBay	✓	×	△	✓	△	✓	△	✓	×	×
Twitter	×	×	△	✓	×	×	✓	✓	×	×
Vehicular Apps	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert
GasBuddy	×	×	✓	✓	×	×	✓	✓	×	×
Google Navigation	✓	✓	×	×	△	×	✓	✓	✓	✓
TomTom Navigation	×	×	×	×	✓	✓	✓	✓	×	×
Waze	✓	✓	△	×	×	×	△	✓	×	×
iRadar	×	×	×	×	×	×	✓	✓	✓	✓
Beat the Traffic	✓	✓	△	✓	×	×	△	✓	✓	✓
Best Parking	✓	✓	✓	✓	×	×	✓	✓	×	×
Industry App	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert	AMC	Expert
MPG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

This table compares AMC's results against those produced by manual testing by an industry expert. For each combination of application and usability property, the table shows AMC's result and the expert's result. A × indicates a violation, a ✓ indicates no violation, and a △ shows that AMC was not able to make a definitive judgment.

Table 3.1: Comparison of AMC testing with manual expert testing

Table 3.1 compares the results of AMC and manual expert testing. For each form of testing, a check indicates no violation, and an x indicates that the application violates the best practice guideline. A triangle in the AMC column shows that AMC could not determine on its own whether the application violated the user interface guideline. In these cases, one or more screens are flagged for subsequent manual analysis.

AMC reached a definitive judgment on 85% of the application/property combinations. Over 98% of these judgments were correct; i.e., they matched the ground truth expert assessment.

AMC had one false negative (word count for eBay). In this case, AMC was not able to visit the state that had the violation because of its state equivalence heuristic. The state is only reachable via multiple sub-menus that unfortunately all have the same structure. AMC incorrectly concludes that the sub-menu states are the same, and so it prematurely halts exploration of the sub-menu structure.

AMC cannot render a definitive judgment in 15% of the cases. Three of these cases are animations. Because this property is hard to assess, AMC defers any reasonably sized artifact for manual analysis. While the expert decided that all cases were not violations, AMC still substantially reduced the amount of manual analysis required. There are 381 total states discovered by AMC in the 12 applications. Of these, 49 states contain some form of animation, and AMC flagged only 4 of those states for manual analysis.

The remaining cases for which AMC cannot render a definitive judgment involve clickable regions in custom views. In these cases, AMC determined that view changed after clicking on a region, thereby rendering subsequent analysis useless. Again, although AMC cannot always provide a definitive judgment, it only flags a small set of states for manual analysis. There are 97 custom views in the 381 total states; AMC flagged only 13 of these for manual analysis. Across all five properties, AMC flagged less than 5% of application screens for manual analysis.

Although I did not compare results for 2 AMC properties, I believe that results for those properties will be similar to or better than the results for the other 5 properties. The violation criteria for both properties can be expressed quantitatively, and therefore AMC would not have to defer any checks for subsequent manual analysis.

Application	Runtime	# of states	States found by AMC	By random walk
Popular Apps				
Gmail	0:56:16	30	27	26
Google Finance	0:54:59	20	18	10
MyDays	1:55:03	41	26	15
eBay	0:50:29	29	24	19
Twitter	10:25:25	82	65	53
Vehicular Apps				
GasBuddy	4:28:06	67	59	50
Google Maps	1:59:26	47	31	27
Google Navigation	0:36:48	24	15	14
TomTom Navigation	4:05:13	50	36	20
Waze	6:09:15	98	81	70
iRadar	1:04:56	23	23	22
Beat the Traffic	0:19:02	18	18	16
Best Parking	0:10:26	15	11	8
Industry App				
MPG	1:22:31	5	5	4

Table 3.2: Time to evaluate applications and the number of states explored

3.3.2.1 Microbenchmarks

Table 3.2 shows more information about AMC's state exploration for the 14 programs. The second column shows AMC runtime. The median runtime is approximately 73 minutes, and the average runtime is slightly more than 2 hour and 33 minutes. There is clearly great variation in the time needed to explore an application, with the fastest application (Best Parking) taking about 10 minutes and the slowest (Twitter) taking a little over 10 hours.

Many factors contribute to exploration time. More complex applications that have more states and transitions naturally take longer to explore. The amount of seemingly non-deterministic behavior in an application has a large effect on exploration time. If AMC has difficulty reaching a particular target state with an unexplored transition, it may need to try many different routes before arriving at the target. Finally, applications with many custom views have longer exploration times because AMC must send many events to each custom view to discover the size and number of clickable regions within the view.

I believe AMC's exploration times are already quite reasonable since the tool executes offline, i.e., it requires no human supervision while it runs. Much of AMC's time is spent extracting the DOM tree for each state after a transition; this is a slow process on the older Nexus One phone, but it would run considerably faster on a modern phone. Additionally, exploration is potentially a highly parallelizable task. One can test multiple applications in parallel and potentially even test the same application on different computers with some minimal coordination among the test platforms.

The last three columns of Table 3.2 show the actual number of states in each application (which I computed by hand), the total number of states visited by AMC, and the total number of states that are visited by a purely random walk through the application. For the random walk, I configured the tool to select a random user action on each screen, wait for a transition, and then repeat. I terminated the random walk after it had been running for the exact same amount of time used by AMC for exploration. Since the random walk strategy can not easily handle custom views, I omitted checking such views for the purposes of this comparison.

For every application, AMC explores more states than the random walk strategy. However, AMC only visits 80% of the total states (as opposed to 65% for random walk). Some states are configuration-dependent: they can only be triggered by toggling application-specific menu items in the correct order. For instance, the MyDays application supports 7 different languages, but AMC explores only 2 of these languages (because of its heuristic of clicking on only 2 list items unless a significant difference in behavior is detected). It is possible, but unlikely, that the application will violate a user interface guideline in one language but not another; if so, AMC would fail to detect this.

For such applications, there exists a tradeoff between coverage and exploration time. Relaxing some of AMC's heuristics would lead to a larger number of states explored, but would likely give a poor return on the time spent in terms of finding new violations.

	User actions per task	Text contrast ratio	Text word count	Button size	Button distance	Animation	Scrolling
Popular Apps							
Gmail	17	2	2	0	0	0	3
Google Finance	0	1	3	1	4	0	3
MyDays	20	4	2	0	2	0	1
eBay	0	1	1	0	0	0	2
Twitter	24	28	7	0	14	0	9
Vehicular Apps							
GasBuddy	12	14	1	0	13	0	4
Google Maps	8	7	3	0	7	0	1
Google Navigation	0	0	0	2	1	0	0
TomTom Navigation	26	0	6	1	0	0	6
Waze	72	15	0	1	5	0	4
iRadar	0	2	1	5	4	0	0
Beat the Traffic	5	7	0	0	1	0	0
Best Parking	0	0	0	0	1	0	1
Industry App							
MPG	0	0	0	0	0	0	0

Table 3.3: Number of violations found for each application

3.3.2.2 Assessment of current Android applications

After establishing that AMC is an effective tool for measuring vehicular user interface properties, I used AMC to assess the user interfaces of the 14 Android applications described in Section 3.3.1. Table 3.3 shows the total number of each type of violation detected for these applications. In the rare cases where the assessment of AMC and the industry expert differed, I use the expert's opinion after viewing the AMC results as the definitive judgment as to whether or not a violation exists.

Unsurprisingly, the results show that applications that are not designed for vehicular environments frequently violate the best practice design principles for which I tested. Since one would naturally expect the total number of violations to increase with application complexity, I normalize results by dividing the total number of violations by the number of unique application states. By this metric, the popular non-vehicular applications average 0.75 violations per state.

When I began the study, I expected the vehicular applications to more closely follow best practice guidelines. However, the 8 vehicular applications in the Android marketplace are only slightly better at following best practice guidelines and have a similar ratio of violations to total states (0.69).

The only application for which I found no violations was MPG. This application was developed with cooperation from Ford engineers to demonstrate the capabilities of the OpenXC platform. Although the primary developer was not familiar with vehicular user interface design, he likely benefited from feedback from engineers who had previously developed vehicular applications.

The best practice guidelines that were most often violated were user actions per task and text contrast ratio. This is unsurprising since these are the hardest two properties to check. Reasoning about the minimum number of actions to perform a task requires thinking about multiple states at once. Checking text contrast ratio requires special tools and/or doing complex calculations. On the other hand, I found no violations of the animation property: all instances of animation that AMC observed were judged by the industry expert to be acceptable.

Overall, these results support my conjecture that current vehicular application developers often make poor user interface design decisions. The most likely reasons for this are that they are unaware of best practices and they lack tools that can measure how their GUIs will perform in vehicular settings. AMC is designed to fill this gap.

3.3.3 Limitations

Since many mobile applications are supported by advertising, I had originally suspected that advertising would lead to many violations. Yet, no application that I studied had advertisements that directly triggered a violation. For instance, I did not encounter any advertisements with animation. Of course, advertisements may reduce the screen area available for application controls and indirectly lead to other violations.

AMC assumes that application behavior is mostly deterministic. If an application displays random pop-ups, for instance, AMC will only observe a violation if a pop-up occurs during exploration. Data retrieved from the cloud can also be non-deterministic. For instance, Waze displays nearby users on a map. If AMC exploration runs in the wee hours of the morning, then no users may be nearby; if exploration runs during rush hour, many users may be shown on the same map. AMC flags the map screen as an instance of animation that needs human judgment in the latter case, but not in the former case.

AMC exploration currently only uses GUI buttons and Android built-in actions (e.g., the back button). Therefore, it does not yet find states that can only be reached by performing complex actions such as multi-touch.

Despite of its limitations, AMC has successfully identified many best practice UI guideline violations in vehicular Android applications. By detecting these violations, AMC enables developers to improve the usability of their applications by fixing reported violations.

CHAPTER IV

Scheduling Application-Initiated Interactions in Vehicular Computing

In the previous chapter, I focused on minimizing distractions from vehicular applications when a driver initiates the interaction. In this chapter, I focus on minimizing the distraction caused by an interaction that is initiated not by the driver, but by the application. In a vehicular setting, the user's primary task is driving, and a driver may have very limited attention to spare for the vehicular application. Consequently, to minimize driver distraction and unsafe interactions, such applications often run in the background and try to interact with the driver only when an interaction will be meaningful. This is a fundamentally different model of interaction than that used by traditional desktop systems. Instead of the user initiating the interaction at a convenient moment, e.g., by opening the application, the application now initiates the interaction, e.g., via a touchscreen notification or by an audio tone from the in-vehicle HMI infotainment system. For instance, a text message application running on Ford SYNC [5] or Volkswagen MIB II [80] initiates an audio-based interaction to the driver when a new message arrives. Cadillac CUE's navigation application initiates a visual pop-up interaction to show an alternative route option when there is slow traffic reported in driver's route [69]. These application-initiated interactions benefit the driver by informing him about relevant new events.

Unfortunately, such interactions can easily turn into distractions if they demand too much of the driver's attention. Since the driver is focusing much of her attention on the road, she only has limited attention for a secondary task. If an interaction is too complex (e.g., showing lots of text or requiring keyboard typing), the driver cannot easily understand or respond to the interaction with this limited attention. To compensate, the driver is forced to divert attention from the road and devote it to the interaction. However, priority matters. Some interactions contain safety-related information and have high priority (e.g., an alert about nearby severe weather). Such interactions should always be allowed. To balance the benefit and distraction caused by application-initiated interactions, an in-vehicle infotainment system should *schedule* interactions to be initiated only when they are high priority or when the driver has enough attention to handle them. Otherwise, an infotainment system should defer interactions until a more suitable time.

Currently, all infotainment systems employ a simple scheduling policy based solely on the high-level interaction type (e.g., voice-based or visual text-based) and whether or not the vehicle is moving. The policy is pre-determined by each vehicle manufacturer. For instance, Chevrolet MYLINK [41], Toyota Entune [110], and Mazda Connect [73] always allow voice-based interactions for new text messages. However, they all disallow text-based interactions that require people to read messages from the infotainment system's screen unless the vehicle is stopped. Furthermore, they do not distinguish between whether vehicle is stopped at a stop sign or stopped in a parking lot and treat them as a same. Later in this chapter, I will show that delivering an interaction when the driver is stopped at a stop sign is not desirable since the driver is devoting much of her cognitive attention to check for incoming cars. These coarse-grained scheduling policies that only consider interaction type and whether or not the vehicle is moving ignore two critical factors: (1) a driver's available attention changes significantly as driving conditions vary, and (2) interactions of the same type can demand substantially different amounts of attention.

All existing policies assume that the driver's available attention is unchanging while the vehicle is in motion. However, numerous studies on driver attention [111, 93, 88, 72] report that a driver's available attention varies significantly as driving conditions change (e.g., due to changes in vehicle speed, traffic volume, and the driver's experience). For instance, Senders et al. [93] report a 38% decrease in a driver's available visual attention as the vehicle speed increases from 30 mph to 60 mph. Patten et al. [88] show that a novice driver needs to devote about 30% more cognitive attention to driving in comparison to an experienced driver. Furthermore, my results in Section 4.4.2 show that a usability expert recommends different scheduling decisions for identical interactions as driving conditions change.

Intuitively, a driver has a very little attention to spare when driving on a curvy, busy road on a snowy day. In contrast, he has much more attention to spare when he is driving on a straight road on a clear day with no traffic. Yet, current infotainment systems do not distinguish between these two different driving scenarios and make identical scheduling decisions. As a driver's available attention fluctuates, the infotainment system's simple scheduling policy can become too aggressive or too conservative. When the driver experiences difficult driving conditions, an infotainment system's scheduling policy will allow interactions that the driver cannot handle (e.g., initiating a voice-based text message interaction when the driver is on a busy, snowy road).

In contrast, when the driver has spare attention, current infotainment system policies are too conservative since they do not initiate interactions that the driver can handle. At first, it may appear that conservative scheduling policies are always the safest approach. However, if most interactions are disallowed, the driver will seek alternative, riskier approaches to obtain new information (e.g., using a smartphone to check for new text messages). According to a recent AT&T survey of driver smarthpone usage [10], 62% of drivers keep their smartphones in an easily reachable location, and 70% of them engage in smartphone activities (e.g., checking new text message, e-mail, and social media posts) while driving. Participants of the survey claimed habit, the fear of missing something important, and the belief that both driving and smartphone interaction can be done safely as the three main reasons for why they interact with smartphones while driving. The survey reveals an addiction to new information and a willingness to obtain that information even at the cost of safety (i.e., interacting with smartphone applications). In comments to the National Highway Traffic Safety Administration, the Alliance of Automobile Manufacturers, which consists of 12 vehicle manufacturers, stated “Consumers have numerous connectivity options, particularly via portable electronic devices. They will quickly migrate to alternate, and potentially more distracting and less safe, means of staying connected if the use of in-vehicle or integrated options is overly curtailed.” [81]. In other words, drivers have a strong desire to obtain new information using any means. Conservative scheduling policies cause drivers to bypass the infotainment system and use alternative devices such as smartphones. A better approach is to initiate interactions when the driver has enough spare attention to handle them. The driver can trust the infotainment systems to promptly and safely deliver new information in vehicle-appropriate form factors (e.g., using a dashboard screen and steering-wheel controls). As a result, the driver will be less tempted to seek information using unsafe, alternative methods.

The second problem with existing scheduling algorithms is that they assume all interactions of the same type (e.g., voice-based or visual pop-up based) require the same amount of driver attention. However, the attention demand of each interaction differs significantly based on content. For example, the driver will take much longer to understand a visual pop-up interaction that contains many buttons and lots of words in comparison to the same pop-up interaction with one button and few words. Consequently, there are many detailed guidelines and standards for assessing an interaction's content (UI composition) and minimizing its attention demand in a vehicular setting [29, 45, 77, 97, 104]. Unfortunately, existing scheduling algorithms simplify the attention demand by ignoring interaction content and only using broad classifications of interactions (e.g., whether an interaction is visual or audio-based). As a result of this simplification, infotainment systems can either under-estimate or over-estimate each interaction's actual attention demand. Instead, they should analyze and predict the attention demanded by each interaction individually.

In this chapter, I introduce *Gremlin*, which schedules the delivery of application-initiated interactions in a vehicular setting by dynamically considering a driver's available attention, the attention demanded by each interaction, and the priority of each interaction. As an application tries to initiate an interaction, Gremlin first checks if the interaction is safety-related and has high priority. If so, Gremlin allows it immediately. Otherwise, Gremlin estimates the driver's available attention based on current driving conditions and predicts the attention that will be demanded by the interaction. If the interaction's attention demand does not overload the driver's available attention, Gremlin allows it. Otherwise, Gremlin defers the interaction until a later time when the driver has enough attention.

To make a dynamic decision about whether to allow or defer the scheduling of an interaction, Gremlin needs to determine a driver's available attention and the attention demanded by the interaction. To estimate a driver's available attention, Gremlin constantly monitors driving conditions (e.g., vehicle speed, traffic conditions, and road curvature) and computes how much of the driver's attention can be safely used for a secondary task. Estimating how much attention will be demanded by an interaction is harder, since Gremlin cannot determine the exact I/O that will occur during an interaction and the corresponding attention demand until it allows the interaction. To solve this problem, Gremlin observes past interactions and builds an attention demand prediction model by observing and quantifying attention demanded by low-level

I/O components such as pressing a button, reading text, and listening to audio. This prediction model is derived from past studies [123, 115, 100, 98, 99, 87] that quantify the attention demand as the time needed for a user to fully understand and interact with individual I/O components. Analyzing each component of an interaction to determine its attention demand is computationally expensive. If the analysis is done online, it will significantly slow down the performance and the responsiveness of an interaction and negatively impact the user experience. Instead, for each interaction, Gremlin records the video and audio input/output of the interaction as it happens, and analyzes the recorded data offline on a server to predict the attention demand of future interactions. This is a continuous process; as Gremlin allows new interactions, it learns more about each interaction so that it can make better predictions for future interactions.

In contrast to current scheduling algorithms that are based on qualitative assessments and coarse-grained categories, Gremlin takes a quantitative approach to scheduling interactions. Gremlin regards scheduling of application-initiated interactions as a real-time scheduling problem. In real-time scheduling, each job has a *priority* and *completion time*, and a job is scheduled only if its completion time can occur before its *deadline*. In a vehicular setting, the deadline is the amount of time that the driver can safely spend on a secondary task without negatively affecting his driving. Following numerous psychology and distracted driver studies [24, 124, 115, 68, 111, 88, 98], Gremlin analyzes two dimensions of attention that are most critical in driving: visual and cognitive attention. For each dimension, Gremlin computes a separate deadline. Gremlin denotes the *visual deadline* as the time that the driver can safely look away from the road and the *cognitive deadline* as the time that the driver can safely spend for a non-driving cognitive task (e.g., listening and understanding a voice message). These values are derived from both laboratory studies of how different driving contexts affect driver attention [24, 111, 88, 98] and real-life crash data [121].

Given these deadlines, Gremlin determines if a proposed interaction can be scheduled based on its priority and predicted completion time. Most application-initiated interactions have lower priority than driving and should only be scheduled if they can be serviced within the deadline. To determine if the driver can complete an interaction within the deadline, Gremlin analyzes each step of an interaction. Interactions are usually composed of a series of steps, such as a series of visual screens or a series of audio-based exchanges between the driver and the system. If any step of an interaction cannot be completed within the deadline, the interaction will overload the driver and negatively affect his driving. Therefore, Gremlin quantifies the attention demand imposed by an interaction as the maximum time required for any step; these values are the *visual completion time* and the *cognitive completion time*. Gremlin only schedules an interaction if the visual completion time and cognitive completion time are less than the corresponding deadlines.

In this chapter, I make the following contributions:

- I introduce a technique for quantifying a driver's available attention as driving conditions change based on laboratory studies and real-world crash data.

- I introduce a technique for fine-grained estimation of the attention demand imposed by an interaction based on observed content rather than coarse-grained categorization.

- I show that a simple real-time scheduling algorithm can be applied to application-initiated interactions in vehicles.

I evaluate Gremlin by comparing its decisions to the decisions recommended by an industry expert for 7 interactions in 3 different driving scenarios. The results of this comparison show that Gremlin's decisions match the expert's recommendations over

90% of the time. Further, I show that Gremlin's recording of interactions, analyzing of interactions, and scheduling impose very little overhead on the system.

4.1 Design considerations

There are three important design considerations for Gremlin. The first is defining an *interaction* that Gremlin is scheduling. The second is deciding where to implement Gremlin's real-time scheduling of application-initiated interactions. The last design consideration is determining how to model user attention as a quantified resource that can be used to specify real-time scheduling constraints.

4.1.1 Defining an interaction

Gremlin defines an *interaction* to be a related sequence of user inputs and outputs that corresponds to performing a single logical task. Typically, the interaction will start with a *notification*; this often will be an audio tone, e.g., the one used by Android's default `NotificationManager`. However, some applications use custom notifications, in which case the notification could be a spoken sentence, a visual pop-up, or some combination of the above. The driver might respond to the notification and interact with the application via audio and voice commands, or the interaction may take the form of reading information from the vehicle touchscreen and responding with button presses or haptic steering wheel controls. For interactions that comprise multiple steps, the driver may pause between steps to look at the road and drive the vehicle. The interaction ends when the logical task is complete; this typically happens when the notification is dismissed or the application returns to a home screen.

Some prior systems have scheduled mobile notifications. In contrast, Gremlin is designed to schedule the entire interactions that those notifications initiate. In a complex driving situation, a user may have sufficient attention available to respond to an audio tone (the notification). However, that response may initiate a more complex interaction (listening and responding to a text message), for which the driver cannot spare attention at the moment. This situation is undesirable. The driver may be implicitly led to perform an activity for which attention cannot be spared, causing distracted driving. Alternatively, the driver must decide to abort or pause the interaction upon realizing that it will be inappropriate; unfortunately, simply remembering that a task is pending places a cognitive burden on the user. I believe it is far better to only deliver a notification when the driver is able to perform the interaction that the notification initiates.

4.1.2 Scheduling application-initiated interactions in the OS

There are at least two design options in deciding where to schedule application-initiated interactions. Scheduling can be performed by each application or by the operating system of an infotainment system. Both approaches have their own advantage and disadvantages.

If scheduling is done in the application, the application can obtain information about the current driving situation by requesting various in-vehicle sensor data (e.g., speed and GPS location) from the infotainment system (e.g., Ford's OpenXC platform allows applications to access vehicle sensor data). The application can then estimate the driver's available attention. The application has the best knowledge about its own interaction content and the required driver responses. Therefore, the application is also well-suited to accurately estimate the attention demand of each interaction. This approach allows the application to adapt its content and modality as a driver's available attention changes (e.g., by initiating a voice-based interaction instead of a visual pop-up interaction when the driver has limited attention). Furthermore, since the scheduling is performed at the application level, no modification needs to be made in the infotainment system, and the driver can benefit as soon as the application gets deployed.

However, there are also substantial disadvantages to this approach. First, estimating a driver's attention supply and interaction's attention demand are difficult tasks and the burden should not fall to application developers who are already devoting their effort to creating an application. Furthermore, in order to prevent distractions caused by interactions, all applications need to have a correct scheduling mechanism. Even if all but one application employs perfect scheduling of interactions, the driver will still be distracted by interactions from the single application with poor scheduling. Additionally, each application is unaware of what others are doing. Therefore, two applications can simultaneously believe that the driver has enough available attention, initiate interactions at the same time, and overload the driver's attention. In contrast, implementing scheduling of all interactions in a single common component, such as the OS, avoids these pitfalls.

Additionally, scheduling at the OS level ensures that most interactions, regardless of their origin applications, will be properly scheduled without overloading the driver. The majority of application-initiated interactions must go through the OS to initiate a user interaction (e.g., as today's mobile phone apps must go through system software to deliver a push notification). Having a common scheduler in the OS significantly reduces the development effort compared to scheduling interactions in every application.

Most importantly, scheduling interactions at the application level jeopardizes driver's privacy. Determining the driver's available attention requires understanding the driver's current driving situation, which in turn requires access to raw vehicle sensor data such as which road the driver is on, the vehicle speed, etc. If each application is responsible for estimating the driver's available attention, then every application would require access to a wealth of private information. A driver would have to trust that each application would not use her private information maliciously. In contrast, if the scheduling is performed by the OS, the driver does not need to worry about such issues, since the OS is already trusted with the privacy of this data.

Unfortunately, this approach is not without drawbacks. The OS has very limited information about an interaction compared to the application itself. Yet, the OS needs to know the detailed contents of an interaction to estimate attention demand before scheduling an interaction. One way to overcome this drawback is for the OS to continuously monitor interactions and learn about their visual and audio compositions as they occur. For interactions that utilize an OS framework to deliver messages to the user (e.g., Android's $NotificationManager$), the OS can monitor interactions easily since it knows the beginning and the end of an interaction. For interactions that do not utilize any OS framework (e.g., a pop-up within the application), I have created an explicit API that applications use to notify the OS about the beginning and the end of an interaction. With this information, the OS can monitor all inputs and outputs that occur during the interaction and analyze them to predict the attention demand of future interactions.

Based on the considerations, I have implemented Gremlin in the operating system. More specifically, Gremlin performs its real-time scheduling in Android's $NotificationManager$ framework, which is responsible for delivering push notifications to the user.

4.1.3 Quantifying attention for real-time scheduling

Historically, user attention has been measured qualitatively. When automotive companies determine which interactions should be allowed in a vehicular setting, usability experts either perform a series of user studies for each interaction or use their judgement to determine whether such interactions would be appropriate. Unfortunately, this qualitative approach requires an assessment of attention demand for every possible interaction, which is time-consuming. Furthermore, it would also require an qualitative assessment of a driver's available attention in different driving conditions in order to be adaptive.

Therefore, Gremlin models attention quantitatively. Quantifying user attention enables Gremlin to use simple real-time scheduling approaches to manage interactions. Gremlin uses a multi-channel attention model suggested by past psychology studies [8, 117]. These studies argue that attention should be viewed as a composition of multiple independent dimensions such as visual, audio, and cognitive rather than as a single value. Furthermore, they state that a person can perform two tasks simultaneously as long as the tasks do not overload any of these individual attention dimensions (e.g., one can simultaneously drive and listen to music). From these studies, I infer that each attention dimension is independent from one other, and that each dimension has its own threshold.

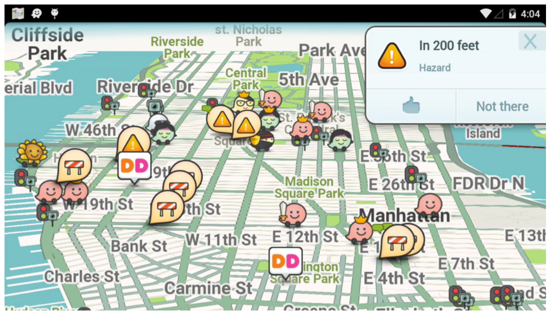
Among the various attention dimensions in a multi-channel attention model, the visual and cognitive dimensions are the most critical for the driving activity, since the driver needs to see the road (using visual attention) and respond (using cognitive attention). Numerous past distracted driver studies [111, 58, 32, 24, 124, 115, 68, 88, 98] have employed this multi-channel attention model and solely focused on a driver's visual and cognitive attention. These studies measured the impact of a certain driving condition (e.g., road curvature or traffic volume) on the driver's visual and cognitive attention separately. Furthermore, these studies utilized time as a unit of measure in quantifying driver's attention. For instance, Tsimhoni et al. [111] determined the effect of road curvature on a driver's visual attention by measuring how long the driver can safely stay occluded (i.e., not having any visible view of a road) as the driver goes through a curve. Following this attention model, Gremlin solely focuses on the driver's visual and cognitive attention and separately quantifies each attention dimension.

Gremlin quantifies a driver's available attention as the time that can be safely spent on a visual or cognitive task before returning to the primary task of driving.

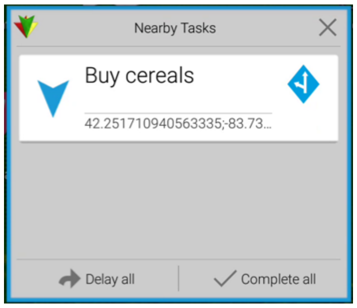
Based on previous laboratory studies that study such factors as vehicle speed, lane width, and road curvature [93, 111, 24], Gremlin quantifies a driver's available visual attention supply as the time the driver can safely look away from the road. Gremlin quantifies a driver's available cognitive attention as the time the driver can safely spend on a secondary cognitive task. This time is determined by utilizing studies on real-life vehicle crash data [121] and studies of cognitive reaction time based on driver experience, traffic volume, and specific situations (such as being stopped at a stop sign) [88, 72].

These values can be viewed as *deadline* constraints in real-time scheduling. Gremlin denotes a driver's available visual attention supply as $V_{deadline}$ and a driver's available cognitive attention supply as $C_{deadline}$. In order for an application-initiated interaction to be safe, the attention demand by the interaction must be less than these deadlines. For instance, when $V_{deadline}$ is 2 seconds, the driver can only safely look away from the road for 2 seconds. Therefore, any interaction that requires more than 2 seconds of visual attention should not be allowed. In Gremlin's real-time scheduling, each interaction's attention demand is viewed as a required *completion time*. Gremlin estimates how long it will take for the driver to visually/cognitively understand and respond to an interaction and only schedules an interaction if visual and cognitive completion times are less than the corresponding deadline.

Gremlin denotes the visual attention demand of an interaction as $V_{completion\_time}$. For example, $V_{completion\_time}$ can be estimated as the sum of the time needed for a driver to read the text on the screen (based on word count and contrast ratios) and touch a response button, which requires the driver to direct his visual attention to a touchscreen. Gremlin defines the cognitive attention demand of an interaction as $C_{completion\_time}$; this is estimated by measuring additional delay in driver reaction time as a result of performing secondary tasks [109, 83, 98, 99]. For example, this captures the cognitive demand imposed by audio-based interactions based on type (sound or



(a) Waze alerting about reported hazard (Considered as high priority by Gremlin)



(b) Geotask alerting when driver is near a store (Considered as normal priority by Gremlin)

Figure 4.1: Examples of interactions that are initiated by an application

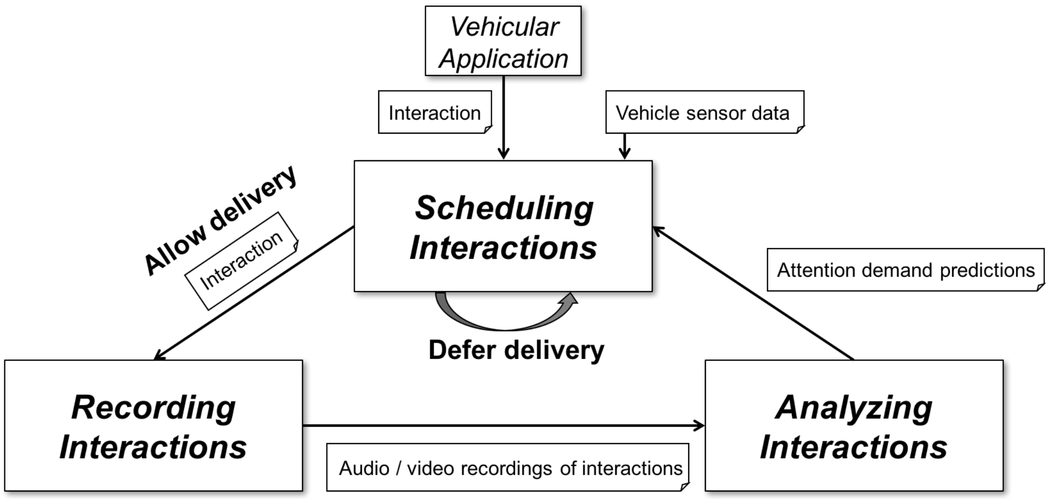


Figure 4.2: Gremlin's three cyclical tasks

speech) and complexity.

4.2 Overview

Gremlin performs real-time scheduling of Android application-initiated interactions to ensure interactions do not distract the driver by overloading his available attention. Examples of Android application-initiated interactions that a driver may encounter are shown in Figure 4.1. When they are scheduled at the right time, these interactions provide useful information to the driver without overloading the driver's attention. But, if they are scheduled at the wrong time, they may become a distraction since the driver is too busy and focused on the road.

As illustrated in Figure 4.2, Gremlin continuously performs three tasks: it records interactions as they occur, it analyzes recorded interactions offline to model their attention demand, and it schedules interactions by either allowing them or deferring them to a more suitable time. Gremlin estimates the driver's current attention level on the device since estimating attention supply is computationally inexpensive. However, determining the attention demand of an interaction requires computationally expensive analysis of individual UI elements on the screen and the audio input and output. Therefore, I have chosen to generate the interaction attention demand model offline.

When the driver uses Gremlin for the first time, Gremlin can immediately estimate the driver's available visual and cognitive attention ($V_{deadline}$ and $C_{deadline}$) based on vehicle speed, road curvature, driver experience, etc. However, it cannot estimate visual and cognitive attention demand imposed by interactions ($V_{completion\_time}$ and $C_{completion\_time}$), since it does not have any attention demand model generated for individual interactions. Therefore, Gremlin initially performs a conservative scheduling based on broad classifications (i.e., audio-based vs. visual-based interactions). As an initial scheduling policy, Gremlin schedules interactions that contain any visual components only when the vehicle is stopped for an extensive period of time. For audio-based interactions, Gremlin initially schedules them either when the vehicle is not moving or moving at low speed. As the driver interacts with the application, Gremlin records and captures a video of the screen output and captures every raw audio input and output that occur during the interaction. This captured data is stored on the device to be analyzed later. Gremlin starts recording at the beginning of an interaction and terminates it when the driver finishes the interaction. Gremlin recognizes the beginning and the end of an interaction from the explicit API calls made from the interaction-initiating application. Gremlin also supports the applications that do not use its API and instead utilize Android's default $NotificationManager$. For these interactions, Gremlin infers about the beginning and the end by determining when the notification call was made to $NotificationManager$ and when the driver either dismisses an interaction or completes the required tasks and leaves the application screen (e.g., goes to a home screen). This recording process imposes minimal overhead and does not affect the responsiveness of the device. Details of how Gremlin records an interaction are described in Section 4.3.1.

In the second phase, Gremlin analyzes these recorded interactions to generate the attention demand model for each interaction. For each recorded interaction, Gremlin estimates its visual attention demand, $V_{completion\_time}$, by analyzing each screen's word count, text contrast ratio, and button sizes. Gremlin computes the cognitive attention demand, $C_{completion\_time}$, by analyzing the content and complexity of each audio input and output. Gremlin builds the attention model by calculating the distribution of past attention demand values for the same interaction. Using this distribution, Gremlin can predict the visual and cognitive attention demand of subsequent interactions. Details of how Gremlin analyzes recorded interactions to predict each interaction's future attention demand are explained in Section 4.3.2.

Given the model of attention demand, Gremlin performs better scheduling of application-initiated interactions. When an application initiates an interaction, Gremlin predicts its attention demand and compares that value to the driver's available attention. Gremlin schedules an interaction when both $V_{deadline}$ and $C_{deadline}$ are greater than $V_{completion\_time}$ and $C_{completion\_time}$. Otherwise, it monitors changes in the driver's available attention supply, and schedules the interaction once the driver's $V_{deadline}$ and $C_{deadline}$ become larger than $V_{completion\_time}$ and $C_{completion\_time}$.

Gremlin enforces the above scheduling policy for the vast majority of interactions. However, certain interactions, such as alerting a driver about an immediate road hazard as in Figure 4.1a, are high priority and should be delivered to the driver right away because they contain critical safety-related information. Gremlin initiates high-priority interactions regardless of the driver's available attention level. Details of how Gremlin estimates driver's available attention and performs scheduling are explained in Section 4.3.3

As Gremlin schedules an interaction, these three tasks are repeated. Gremlin will record the newly scheduled interaction, analyze it to improve the attention model, and make better scheduling decision when it encounters the same interaction in the future. By continuously performing these three tasks, Gremlin improves its scheduling in a fully automated fashion without the need for manual intervention.

4.3 Design and implementation

Gremlin schedules application-initiated interactions without overloading the driver's available attention. To achieve its goal, Gremlin continuously performs tasks in three phases: recording interactions, analyzing interactions, and scheduling interactions. Gremlin's two tasks of recoding and analyzing interactions are designed to work together to generate the attention demand model of interactions. Gremlin later utilizes this model in its scheduling interactions phase to predict the attention demand of interactions that needs to be scheduled and make a scheduling decision by comparing the demand to the driver's estimated available attention supply. In this section, I illustrate the details of how each component works.

4.3.1 Recording interactions

In order to estimate a interaction's visual attention demand, $V_{completion\_time}$, and cognitive attention demand, $C_{completion\_time}$, Gremlin needs to first learn about the composition of an interaction and the types of actions that the driver can perform during an interaction. An interaction may consist of multiple screens that require multiple button clicks, or it may consist of a single screen with a dismiss button. For an audio-based interaction, it may consist of a sequence of audio inputs and outputs or just a single audio output. As a result, Gremlin needs a mechanism to uncover the mystery behind each interaction. Gremlin could perform an AMC-like [67] automated exploration on an interaction to discover its content. However, it is challenging to perform such exploration since the interaction often gets initiated by an external stimulus (e.g., new content on the server or new traffic information) and it is difficult to simulate such external stimulus in an automated fashion.

As a result, Gremlin learns about an interaction by recording the driver's past actions and analyzing them later. At first, Gremlin's recording task is in an idle state: simply waiting for an application to initiate an interaction. Once Gremlin schedules an interaction, it begins recording all the inputs and outputs until the end of the interaction. More specifically, Gremlin captures the video and audio input/output of an interaction. Video of interaction is needed to later analyze $V_{completion\_time}$, the visual demand of an interaction. Audio recordings are utilized to analyze $C_{completion\_time}$, the cognitive demand of an interaction. For each interaction recording, Gremlin stores its *interaction ID*, which is composed of the initiating application's package name as well as the interaction's name and id that are given by the application. The interaction ID is used later to group interactions during Gremlin's interaction analysis phase. When the interaction has been completed by the driver, Gremlin stops the recording task and waits for the next interaction to record. Overall, Gremlin's recording task imposes non-perceptible overhead on the system.

When capturing the video of an interaction, Gremlin utilizes Android's *screenrecord* binary, which records the Android device's screen content into a video file. By capturing to a video file instead of performing multiple screenshots, Gremlin minimizes the number of disk I/O operations and leverages a video encoding mechanism to reduce the size of the captured file. In addition to the graphical content of the screen, Gremlin also captures information about the GUI composition of each screen.

GUI composition information is needed to extract the button size, text, and text color information from the captured video screen. Gremlin later utilizes this information to analyze the visual demand of an interaction. I have modified Android's $ViewServer$ and $HierarchyViewer$ service to extract GUI information of the current screen every second and output it to a file. Using timestamp information, Gremlin can later match extracted GUI information to the exact screen in the captured video and estimate the number of seconds needed to visually understand each screen. I assume that each screen will last at least one second and any screen that lasts less than one second is a negligible intermediate screen (e.g., a loading screen or transition screen).

Gremlin records audio output to Android's *AudioFlinger* service, a low-level service that puts all audio data into the speaker's hardware buffer. As each audio output occurs, my modified *AudioFlinger* captures audio data into a file. When capturing the audio input of an interaction (e.g., a voice command from the driver), Gremlin captures it in Android framework's *AudioRecord* object. Since *AudioRecord* object is the only API that Android provides for obtaining audio input, I have decided to record audio input at the Android framework level.

One of the most important tasks that Gremlin needs to perform during this recording phase is determining when to start the recording and when to end it. Ideally, Gremlin needs to start recording the video and audio content of an interaction at the beginning of an interaction and end once the interaction is complete. In Android, there are two ways an application can initiate an interaction. The first method is using Android's default notification system, $NotificationManager$. The application can initiate an interaction by calling $NotificationManager$'s `notify()` method. Then, Android delivers the interaction content to the driver by displaying a visual pop-up accompanied by an audible ding. To capture the beginning of such interaction, I have modified Android's $NotificationManager$ to allow Gremlin to begin its recording task whenever the application calls $NotificationManager$'s `notify()` method. Gremlin ends the recording task when the driver either dismisses an interaction or leaves the application's screen (e.g., by going back to a home screen). Gremlin recognizes that the driver has dismissed an interaction by monitoring $NotificationManger$'s `cancel()` method.

Another way for an application to initiate an interaction is to use application-specific custom methods (e.g., Waze alerts driver using its own API). Since Gremlin is running on the Android framework, it is unaware of the internals of an application. Furthermore, Gremlin can not control the scheduling of such interactions. Therefore, to support such interactions, Gremlin provides an API for application-specific custom interactions to mark the beginning and the end of an interaction. With the help of applications, Gremlin can successfully record custom interactions and later analyze and schedule these interactions as well. Details of Gremlin's API are shown in Section 4.3.3.4.

4.3.2 Analyzing interactions

After Gremlin records interactions, it analyzes them offline. Gremlin's offline interaction analyzer tool is developed in Java. In the current implementation, Gremlin requires the manual transfer of recorded interaction data from an Android device to its offline analyzer. However, this manual process can be eliminated by transferring recorded interaction data via wireless network as a background process in the Android device.

When recorded interaction data is successfully transferred to the analyzer, Gremlin begins to estimate each captured interaction's visual and cognitive attention demand, denoted as $V_{completion\_time}$ and $C_{completion\_time}$. To compute $V_{completion\_time}$, Gremlin estimates the number of seconds that would take for a driver to understand and respond to individual screens based on the number of words, text contrast ratios, and button sizes. After computing the visual attention demand of every recorded screen, Gremlin finds the maximum attention demand and denotes it as the $V_{completion\_time}$ of an interaction. The reason behind for this maximum approach is that an interaction should not be initiated if the driver does not have enough visual attention to handle the screen that requires the most visual attention. Gremlin follows the same logic when computing $C_{completion\_time}$ of an interaction. For each audio input and output, Gremlin analyzes its type (speech or non-speech) and speech complexity. Based on these factors, Gremlin estimates the needed cognitive attention demand for each audio interaction and denotes the maximum cognitive attention demand as $C_{completion\_time}$ of an interaction. Gremlin does not compute $C_{completion\_time}$ based on visual elements of an interaction, since $V_{completion\_time}$ for such elements will be much higher than $C_{completion\_time}$, and it will be the dominant factor in scheduling.

After estimating $V_{completion\_time}$ and $C_{completion\_time}$ of each recorded interaction, Gremlin combines past $V_{completion\_time}$ and $C_{completion\_time}$ values of the same interactions (identified by an interaction ID) to build individual $V_{completion\_time}$ and $C_{completion\_time}$ distributions. These distributions illustrate the variance of attention demand in the same interaction. If an interaction portrays the same content every time it is delivered, its $V_{completion\_time}$ and $C_{completion\_time}$ distribution will always be the same. However, for most interactions, the attention demand changes as the delivered content changes. For instance, a pop-up that displays the text of incoming messages will impose different attention demand on the driver as the messages change. In the current implementation, Gremlin computes 95% confidence intervals for each interaction's $V_{completion\_time}$ and $C_{completion\_time}$ assuming a normal distribution. I assume the attention demand of an interaction will follow a normal distribution, since the composition of an interaction (e.g., having a response button, pop-up that display text) is typically constant and only content changes (e.g., different incoming messages) affects the driver's attention. After computing the 95% confidence interval for each interaction's $V_{completion\_time}$ and $C_{completion\_time}$, Gremlin transfers these values to the device so that they can be used by Gremlin when it makes scheduling decisions.

In the next subsections, I explain in detail how Gremlin computes $V_{completion\_time}$ and $C_{completion\_time}$ of each interaction from the recordings.

4.3.2.1 Computing $V_{completion\_time}$

To compute $V_{completion\_time}$ of a screen, Gremlin's offline interaction analyzer first needs to match screens from the recorded video to the GUI composition information that was recorded every second. For each GUI composition dump, the analyzer finds the corresponding screen clip in the video by matching the timestamp. At the end of this matching process, the analyzer generates the series of screenshots along with their corresponding GUI dump information (e.g., button size and text color information). The screen clip is utilized to extract the color information that are not show in the GUI composition dump (e.g., background color of text).

Next, the analyzer computes each screen's visual demand by analyzing two visual components: text and buttons. To understand the content of a screen, the driver has to read the text presented on the screen; reading this text consumes driver's visual attention. Similarly, a screen may require the driver to respond by touching a button. Touching the button requires visual attention, since the driver has to focus his visual attention on the screen to check the location of the button and ensure that he is touching the right button. Using these two components, the visual demand of a screen is defined as the sum of the time required to understand the text and the time needed for the driver to touch the response button on the screen. After computing the visual demand (i.e., required visual attention time) of an interaction's individual screens, Gremlin's analyzer determines the maximum visual attention among them and denotes it as the $V_{completion\_time}$ of an interaction.

To determine the number of seconds needed for the driver to read the text presented on a screen, Gremlin's offline analyzer utilizes the word count and text contrast

Contrast ratio	$M_{contrast}$
Baseline (black-on-white)	1
3:1	1.125
5:1	0.825
8:1	0.813
10:1	0.888

Table 4.1: $M_{contrast}$, a multiplier value to baseline reading time based on different contrast ratio

ratio of a screen. An average adult can read about 300 words per minute in normal conditions [123]. With this information and the word count of an interaction, the analyzer can estimate baseline time that user will need to read the text. However, the time required to read text also depends on the contrast ratio of text, the background color, font size, font spacing, etc. The analyzer focuses on the impact of the contrast ratio, since many previous studies have highlighted the importance of contrast ratio on visual performance [124, 115, 68]. But, other factors can be later added to improve Gremlin's visual attention demand model.

The study performed by Wang et al. reports an increase in visual performance as the contrast ratio increases from 2:1 to 8:1 [115]. Furthermore, the study illustrates that visual performance does not increase as the contrast ratio passes 8:1. By utilizing the results of this study, I have created a contrast ratio multiplier, $M_{contrast}$, that needs to be applied to the baseline reading time based on the contrast ratio of the text. More favorable reading conditions will lower $M_{contrast}$. When the text contrast ratio is either below 3:1 or above 10:1, Gremlin's offline analyzer sets $V_{completion\_time}$ to a maximum value so that such interaction will not be delivered to the driver to conform with safety guideline [97]. $M_{contrast}$ values for the contrast ratio between 3:1 and 10:1 are shown in Table 4.1. The visual demand for contrast ratio 4:1, 6:1, 7:1, 9:1 was not computed in the study performed in [115]. Therefore, for these ratios, I estimate $M_{contrast}$ by linear regression.

Button size (covered area)	Estimated reaction time (seconds)
20*20 (400)	1.666s
30*30 (900)	1.321s
40*40 (1600)	1.134s
50*50 (2500)	1.168s

Table 4.2: Estimated reaction time (in seconds) for touching a button based on various size in pixels.

In addition to determining visual demand imposed by the text, the analyzer estimates the visual demand of buttons. More specifically, it estimates the time that is needed for the driver to touch a response button. The study performed by Sun et al. measures the reaction time of touching one buttion in a displayed set of buttons [100]. The study used different sets of buttons that varied in size, spacing, and content (whether a set contain only digit buttons, icon buttons, or both). The study illustrates that the reaction time of touching a button becomes slower as button size decreases and when there are mix of digit and icon buttons presented on the screen. Utilizing this study, Gremlin's offline analyzer estimates reaction time for touching buttons of various sizes. Estimated reaction times are shown in Table 4.2

Given an interaction, the analyzer will assess the size of buttons that are presented on the screen. Unlike the controlled study performed by Sun et al., where a mobile touch screen consisted of buttons of all the same sizes, most screens have buttons that vary in sizes. In this case, Gremlin's interaction analyzer will compute the average size of buttons and uses the average button size to determine the estimated reaction time as specified in Table 4.2. For button sizes that are not listed in the table, the analyzer relies on liner regression to estimate the reaction time.

Here is a simple example of how the above components work together in the system. Suppose there is an interaction with one screen that contains 5 words with contrast ratio of 5:1 and a response button with a size of 40*40 pixels. Gremlin's offline analyzer first estimates the baseline reading time to 1 second. Given $M_{contrast}$

of 0.825, the visual demand of the text is 0.825 seconds. Additionally, clicking the 40*40 button requires visual attention of about 1.1 seconds. Therefore, the analyzer concludes that $V_{completion\_time}$ of this screen is 1.925 seconds (0.825 + 1.1).

4.3.2.2 Computing $C_{completion\_time}$

Similar to the visual attention demand model, Gremlin quantifies the cognitive attention demand of an interaction, $C_{completion\_time}$, as the number of seconds that driver needs to spend to cognitively understand and respond to an interaction. Unlike visual attention, cognitive attention is difficult quantify directly since I cannot look inside of a person's brain and determine how much attention is being demanded by an interaction. As a result, Gremlin relies on studies that estimate cognitive workload of the driver indirectly. One common method is measuring peripheral detection task (PDT) response time [88, 47, 57]. Typically, these studies initiate a PDT signal from the peripheral vision of a driver at every 3-5 seconds. Then, the studies measure how long it takes for a driver to notice this signal. As a driver experiences higher cognitive workload, the PDT response time increases. For instance, the average PDT response time when the driver is talking to a passenger is 900ms, which is about 29% higher than with a baseline of undistracted driving [98]. Therefore, it can be concluded that 200ms additional PDT response time is the cognitive attention demand of talking to a passenger. Gremlin denotes the additional PDT response time as $C_{completion\_time}$.

Since I assume that the cognitive demand of a visually intensive interaction is not relevant to the scheduling decision, Gremlin's offline analyzer only computes the cognitive demand of an audio-based interactions. To determine $C_{completion\_time}$ of an audio-based interaction, Gremlin utilizes studies performed by Strayer et al. [98, 99]. Strayer et al. have performed extensive research in quantifying cognitive demand of various audio-based in-vehicle interactions (e.g., giving a voice-based command or listening to a radio). Although these studies illustrates the increase in PDT response time due to various audio-based interactions, these values cannot be used directly since each study's baseline PDT response time differs. Furthermore, other studies [88, 72] that Gremlin utilizes to determine a driver's available cognitive attention supply also have different baseline PDT values. Therefore, I normalize reported PDT values by using the delta from the studies' reported baseline value.

As a first step in estimating $C_{completion\_time}$, Gremlin's interaction analyzer categorizes an audio-based interaction into four categories: listening to non-speech sound only, giving voice commands only, listening to speech only, and listening to speech & replying. To determine if the audio contains a speech or a sound, Gremlin's interaction analyzer utilizes Pocketsphinx [23] for speech recognition. After the categorization, $C_{completion\_time}$ for each categories is computed by incorporating the studies performed by Strayer et al. [98, 99].

For categories that have listening to a speech as a component, the interaction analyzer determines the speech complexity when estimating $C_{completion\_time}$. Patten et al. [87] have measured the PDT response time when a driver is having a complex conversation. By comparing this PDT response time to the PDT response time of having a normal conversation reported in [98], I conclude that listening to complex speech will increase the cognitive load about 48%. As a result, Gremlin's interaction analyzer adds two additional categories to its $C_{completion\_time}$ model. They are listening to complex speech only, and listening complex speech and replying. Gremlin's full $C_{completion\_time}$ model is shown in Table 4.3.

To determine whether a certain speech is complex or not, the interaction analyzer computes the Flesch reading-ease score (FRES) [63] on the content of a speech. In general, if the FRES is 60 or higher, it is considered as plain English. If the FRES is lower than 60, it is considered as more difficult to understand. Gremlin's interaction analyzer incorporates the FRES model when determining if an audio has complex speech or not. If an interaction contains an audio that has a FRES score lower than

Category	$C_{completion\_time}$ (in seconds)
Listening to non-speech sound only	0.004s
Giving voice commands only	0.055s
Listening to normal speech only	0.066s
Listening to complex speech only	0.099s
Listening to normal speech & replying	0.155s
Listening to complex speech & replying	0.230s

Table 4.3: Gremlin's $C_{completion\_time}$ values in seconds (estimated increase in PDT response time)

60, the analyzer will consider such interaction as having a complex speech component.

4.3.3 Live scheduling of application-initiated interactions

In the previous two phases, Gremlin generated an attention demand model that predicts $V_{completion\_time}$ and $C_{completion\_time}$ of future interactions. In this phase, Gremlin estimates the other side of the equation: the driver's available attention supply, denoted by $V_{deadline}$ and $C_{deadline}$.

Gremlin defines $V_{deadline}$ as the number of seconds that the driver can safely take his eyes off the road for a secondary task. $C_{deadline}$ is defined as the number of seconds that the driver can spend for a cognitive task (e.g., thinking and understanding a voice command) without increasing the risk of getting involved in an accident. Ultimately, both $V_{deadline}$ and $C_{deadline}$ provide the allowed budget for a driver to perform a secondary task (e.g., respond to an interaction).

There are many variables that can affect a driver's available visual and cognitive attention. In the current implementation, Gremlin incorporates three factors: vehicle speed, road curvature, and lane width when estimating $V_{deadline}$. To compute $C_{deadline}$, Gremlin utilizes information on driver's experience and surrounding traffic volume. The only information that the driver has to manually provide is his or her driving experience. All other information can be obtained using vehicle's sensor data or online traffic/road information servers. For instance, the vehicle's speed can be obtained directly from vehicle's OBD2 port. To obtain the information about the road curvature, Gremlin can use vehicle's GPS sensor data along with map information from OpenStreetMap service, which specifies the road curvature of the current road [84]. In the current implementation, Gremlin obtains its needed information from a vehicle trace file for a simulated environment study. To accurately capture the changes in a driver's attention, Gremlin continuously computes $V_{deadline}$ and $C_{deadline}$.

In the next subsections (Section 4.3.3.1 & Section 4.3.3.2), I explain how Gremlin estimates $V_{deadline}$ and $C_{deadline}$. After quantifying driver's available visual and cognitive attention, Gremlin performs scheduling as described in Section 4.3.3.3.

4.3.3.1 Computing $V_{deadline}$

$V_{deadline}$ represents how many seconds the driver can safely afford to look away from the road. To estimate $V_{deadline}$, Gremlin utilizes past studies that measure how long a driver can safely stay occluded with changes to vehicle speed, lane width, and road curvature [93, 111, 24]. In these studies, subjects were driving with occlusion glasses in which their view were occluded. A subject can activate a switch when he feels needed the need; this gives him a clear view of the road for 0.5 seconds. These studies measure the time of occlusion for different driving conditions, which shows how long people can stay occluded with 0.5 second exposure of the road until they are forced to activate the switch again for clear view. For instance, a driver must stay occluded for less time when he is entering a sharp curve than when he is driving on a straight road [111]. Since the time of occlusion illustrates how much time user can afford to spend not looking at the road, I interpret it as the driver's available visual attention, $V_{deadline}$.

Past studies [93, 111, 24] that Gremlin relies on measure the impact of one variable at a time on a driver's visual attention. For instance, Senders et al. [93] examine the impact of vehicle's speed on driver's visual attention, and Tsimhoni et al. [111]

examine the impact of road curvature. Fortunately, all of these studies use the same metric in measuring the driver's available visual attention level, which is how many seconds can a driver be occluded after 0.5 seconds of clear view. In order to determine $V_{deadline}$ with consideration to the vehicle's speed, road curvature, and lane width, Gremlin needs a model that can measure impact of these conditions jointly.

One method of creating a such model is to take one study's result directly as a baseline $V_{deadline}$. To incorporate other studies' findings, Gremlin first individually computes the relative change in occlusion time due to the external factor from each study. Then, Gremlin multiplies these relative changes with the baseline $V_{deadline}$ to obtain a $V_{deadline}$ that incorporates all factors jointly. The following equation provides the overview of Gremlin's mechanism for determining $V_{deadline}$.

$$V_{deadline} = V_{deadline}(Speed) * Pe(r) * Pe(w)$$
$$\text{(Pe(r)=Penalty based on the radius of curve, Pe(w)=Penalty based on the lane width)} \quad (4.1)$$

$V_{deadline}(Speed)$ gives available visual attention based solely on the speed of the vehicle. I utilize findings of Senders et al. [93] to determine the baseline $V_{deadline}(Speed)$. Senders et al. measured the impact of speed on the time of occlusion. Using their results, I performed linear regression to obtain following equation.

$$V_{deadline}(Speed) = -0.0408 * speed + 4.0558 \text{(Units in seconds, speed in MPH, } R^2 = 0.9585\text{)} \quad (4.2)$$

From this baseline $V_{deadline}$, I apply percent decrease functions to capture changes in available visual attention caused by road curvature and lane width. Tsimhoni et al. [111] analyzed the impact of the road curvature on the driver's visual attention. Instead of using the time of occlusion as a direct measure of the driver's visual attention, Tsimhoni et al. created their own quantification model, $VisD$, which is simply

0.5 seconds (time of clear view) divided by the time of occlusion. Therefore, I can compute the time of occlusion from $VisD$. I obtain the following formula from the study:

$$\text{Time of occlusion}= \frac{0.5}{0.252+34.5*1/R}$$

$$\text{(Time of occlusion (i.e. } V_{deadline}\text{) as one enters the curve. R=radius of curve in meters)} \quad (4.3)$$

$Pe(r)$ is the relative impact of road curvature on $V_{deadline}$. Here is an example of how Gremlin computes $Pe(r)$. The baseline time of occlusion reported in the study was 1.984 seconds (i.e., driving on a straight road). Let's say a driver is entering a curve with the radius of 196m. Using a Equation 4.3, the time of occlusion decreases to 1.168 seconds. This is a 59% decrease from the reported baseline number. In this example, $Pe(r)$ is 59% and it will be applied to Gremlin's baseline $V_{deadline}$. The following equation gives the $Pe(r)$ formula that Gremlin uses:

$$Pe(r) = \frac{\frac{0.5}{0.252+34.5*1/R}}{1.984}$$

$$\text{(Percent decrease in driver's visual attention due to the road curvature. R=radius of curve in meters)} \quad (4.4)$$

$Pe(w)$ is the lane width peanlty function. It decreases $V_{deadline}$ by 2% as w decreases by one foot from the standard 12 feet [24]. Any lane width that is wider than the standard 12 feet is ignored. The following equation gives the $Pe(w)$ formula that Gremlin uses:

$$Pe(w) = 1 - (0.02 * (12 - L));$$

(Percent decrease in driver's visual attention due to the lane width. L=Lane width in feet) (4.5)

To illustrate the meaning of estimated $V_{deadline}$ more clearly, here is a simple scenario. When a driver is driving in a rural road at 20mph on a straight 12-feet standard lane width road, his $V_{deadline}$ is estimated to be 3.243s using Equation 4.1. This means that the driver can visually interact with an interaction for three seconds without negatively affecting his driving. In contrast, if he is driving at 60mph on a very sharp curve at the radius of 60m, his $V_{deadline}$ drops to 0.49s, which implies that the driver has virtually no time to take his visual attention away from the road.

4.3.3.2 Computing $C_{deadline}$

Gremlin defines the driver's available cognitive attention to be the number of seconds that driver can afford to use for a cognitive task, and it is denoted as $C_{deadline}$. In Section 4.3.2.2, I have quantified cognitive attention using studies that measure PDT response time for various driving conditions [88, 72, 109, 83, 98]. For instance, a PDT response for a novice driver is higher than the experienced driver [88], since the novice driver consumes more cognitive attention when driving than an experienced driver does. Using studies that measure a driver's PDT response time for a various driving conditions, Gremlin can determine how much of the driver's cognitive attention is being used for driving.

Unfortunately, estimating the PDT response time of driver for various driving condition is not enough since it does not specify how higher response times negatively impact driving safety. To be useful for Gremlin, I need to understand how much cognitive attention is available for a secondary tasks. This means that I need to (1) understand how secondary tasks affect PDT response time, and (2) set a threshold value beyond which higher PDT response times are unacceptable for safe driving. I first consider how to set the threshold. One method to do this would be to ask usability experts to choose a value. I believe a better approach is to use actual road safety data to set the threshold.

As a first step of determining the maximum threshold, I have utilized a study that analyzes the impact of various secondary tasks (e.g., dialing a phone, changing a radio station) on crashes or near crashes (CNC) using recorded data from people's actual driving [121]. For each secondary task, the study reports a corresponding odds ratio, which indicates the likelihood of a CNC while performing the secondary task in comparison to driving without performing that secondary task. For instance, the study reports an odds ratio of 1.34 for reading. This means that if you are driving and reading at the same time, you have 1.34x more chance of getting involved in a crash or near crash than if you are not reading. If an odds ratio for a secondary task is higher than 1.0, the study concludes that such task has a positive causal effect on a CNC. Details of how the odds ratio is computed are explained in Appendix VI.

If Gremlin utilizes the odds ratio to quantify driver's cognitive attention, its threshold will be 1.0, since any odds ratio greater than 1.0 represents the increasing risk of one getting involved in a CNC. However, Gremlin quantifies the cognitive attention as 'how many seconds are available for a secondary task' and utilizes PDT response time as the metric. Therefore, Gremlin converts an odds ratio value of 1.0 into a PDT response time to determine the maximum threshold. To perform this conversion, I find four secondary activities, which are changing a radio station, talking on the phone, changing the CD, and dialing a hand-held cell phone, that are both present in the study performed by SAE (Society of Automotive Engineers) [121] and in distracted driver studies that utilize PDT measurements [109, 83, 98]. By performing a linear regression, I determine that an odds ratio of 1.0 is equivalent to a PDT

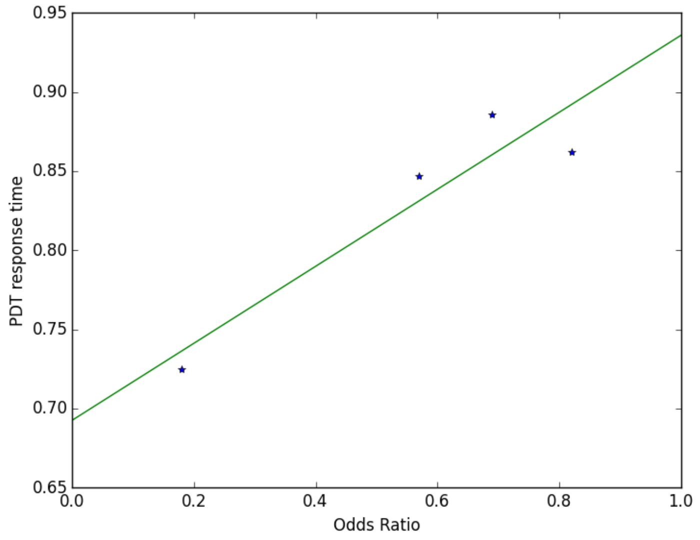


Figure 4.3: The regression between odds ratios and PDT response time for four matching tasks. ($R^2 = 0.875$)

response time about 940ms. The result of the regression is shown in Figure 4.3. This means the driver has total of 0.94 seconds for any cognitive task without negatively impacting her driving.

Since driving itself consumes the driver's cognitive attention, Gremlin subtracts cognitive load imposed by the driving activity from the 940ms threshold. Therefore, the time between 940ms and estimated driver's PDT response time from the driving activity is $C_{deadline}$, the driver's available cognitive attention (shown in Equation 4.6).

$$C_{deadline} = \text{940ms - estimated PDT response time of driving activity} \quad (4.6)$$

Gremlin incorporates the driver's experience and traffic volume when computing $C_{deadline}$. Similar to Gremlin's $V_{deadline}$ model, Gremlin incorporates the results of multiple PDT response time studies [88, 72] into one model. Since these studies have different PDT response time for the baseline driving scenario, I normalize these multiple baselines into one baseline value by taking the average. I further normalize all

	Novice driver	Experienced driver
Low traffic	0.073s	0.274s
Med traffic	0s	0.253s
High traffic	0s	0.152s

Table 4.4: $C_{deadline}$ values (in seconds) based on driver's experience and traffic condition

PDT response time values against this single baseline. The normalized PDT response time for the baseline driving case is 665ms.

First, Gremlin uses previous result from the experiment performed by Patten et al. [88]. After the normalization, estimated PDT response time for a novice driver on low, med, high traffic scenarios are 867ms, 966ms, and 996ms. For more experienced drivers, PDT response times are about 25% lower and they are 666ms, 687ms, 788ms. Applying this estimated PDT response time of the driving activity and 940 ms threshold, Gremlin computes the $C_{deadline}$ as shown in Table 4.4. According to the Table 4.4, there is no room for any additional cognitive task for a novice driver during the medium and high traffic driving scenario.

Additionally, Gremlin incorporates sudden changes of cognitive attention for two spontaneous scenarios, going through a sharp curve and stopped at a stop sign. In these scenarios, the driver will spontaneously devote more cognitive attention to the driving activity. As a result, $C_{deadline}$ will decrease for these scenarios. In Table 4.5, I list two spontaneous scenarios and how much it will decrease the baseline $C_{deadline}$. Each situation and its impact on $C_{deadline}$ are extracted from the study performed by Martens et al [72]. Again, these spontaneous situations are given to Gremlin as a part of the trace. However, Gremlin can utilize the vehicle's GPS position and external map service to determine whether the driver is at a stop sign or going through a sharp curve.

Situation	Changes in $C_{deadline}$ (in seconds)
In a sharp curve	-0.113s
At a stop sign	-0.184s

Table 4.5: Changes in $C_{deadline}$ values (in seconds) for two driving scenarios

4.3.3.3 Scheduling algorithm

When an application initiates an interaction, Gremlin first determines the priority of the interaction. The priority can be either lower priority than driving or higher priority than driving. In the current implementation, the priority of an interaction is either given to Gremlin through its API (discussed in Section 4.3.3.4) or it can be manually assigned (e.g., road safety alerts are assigned as high priority). If an interaction has higher priority than driving, Gremlin delivers this interaction regardless of driver's available attention level.

When the interaction has a lower priority than a driving, Gremlin first checks if its interaction demand table contains the attention demand information about interaction. If this is the first time Gremlin has seen the interaction, the decision maker has no knowledge about interaction's attention demand, so it will deliver only when the vehicle is stopped for a extended period of time (e.g., stopped at a red light) and learn about it via Gremlin's interaction capture tool and interaction analyzer. I believe this is the safer design rather than wildly guessing attention demand of never-seen interaction and cause attention-overloading situation.

In the case where the interaction's attention demand information is present in the interaction demand model, the decision maker will only deliver the interaction if $V_{deadline}$ and $C_{deadline}$ is large enough to cover the upper bound of $V_{completion\_time}$ and $C_{completion\_time}$ interval, which is the upper bound on 95% confidence interval. Gremlin looks at the upper bound, since it employs the conservative approach about its prediction values. If driver does not have enough visual or cognitive attention to handle the interaction, Gremlin will put this interaction in its queue and only deliver when both $V_{deadline}$ and $C_{deadline}$ are large enough to accommodate the demand.

As Gremlin performs its scheduling and deliver an interaction to the driver, Gremlin will continuously capture the inputs and the outputs of an interaction as described in the Section 4.3.1. Gremlin's offline analyzer will later analyze captured traces, update the interaction demand table, and send the latest table back to Gremlin running on Android device. Gremlin will use updated the interaction demand table to perform better scheduling. As a result, Gremlin's three phases continuously work with each other in a cyclic process to better manage precious user attention resource.

4.3.3.4 Using Gremlin

There are two ways for an application to use Gremlin. The first approach is through Android's generic notification using Android's $NotificationManager$ API. Notifications that are initiated using the $NotificationManager$ API get delivered to Android's internal $NotificationManagerService$, where the actual delivery of a notification occurs. Since Gremlin resides in Android's $NotificationManagerService$, any generic notification will go through Gremlin's decision mechanism. Therefore, no changes need be made for applications that use this generic mechanism. If however, an application wants to provide a priority level, it can add priority level as a parameter when calling $NotificationManagerService$. I have modified $NotificationManagerService$ to accept interaction's priority level. If the priority level is not specified, Gremlin assumes the notification has a lower priority than driving.

Some applications utilize a custom notification system to interact with an user. For instance, Waze, a GPS navigation application, initiates an interaction as a pop-up within the application. These type of interactions does not go through Android's $NotificationManagerService$. To support them, Gremlin provides a custom API for an application to use. Unfortunately, Gremlin cannot enforce its decision mechanism

Function	Arguments and return values

`gremlin_begin_notify`	`(IN package name, IN id, IN priority)`
	`-> OUT success`

`gremlin_finish_notify`	`(IN package name, IN id)`
	`-> OUT success`

`gremlin_cancel_notify`	`(IN package name, IN id)`
	`-> OUT success`

Table 4.6: Gremlin's API for custom interactions

for custom notifications, because it cannot control an application's internal operations. Therefore, Gremlin uses its API to guide applications to initiate an interaction when the driver has enough attention. Table 4.6 shows the functions in Gremlin's API for custom interactions.

When the application wants to initiate a custom interaction, it first calls `gremlin_begin_notify`. Gremlin will check if it has estimated attention demand of the interaction. If it does not have any attention demand estimates, Gremlin will allow the interaction returning `success` value with 1. If it does have attention demand estimates, Gremlin will check the driver's available attention and return `success` value with 1 if the driver has enough attention to handle interaction's attention demand. In other words, Gremlin will only return `success` if $V_{deadline}$ and $C_{deadline}$ are bigger than estimated $V_{completion\_time}$ and $C_{completion\_time}$. If not, Gremlin will block on this function call and not return until driver's attention this condition is met.

Since Gremlin has no idea whether the interaction has ended or not, the application needs to explicitly call `gremlin_finish_notify` when the interaction ends. With `gremlin_begin_notify` and `gremlin_finish_notify` calls, Gremlin can know when the interaction starts and ends and use this information during the training phase (using the capture tool and the interaction analyzer).

`gremlin_begin_notify` is a blocking call. Therefore, an application may want to cancel an interaction if it believes that the interaction has become useless. For

Interaction #	Application Name	Priority	Description
#1	Waze	High	Application alerts the driver about the reported hazard ahead. Applications sends the audio output and a visual pop-up.
#2	Waze	Low	Application alerts the driver about the speed camera. Application sends out an audio beep and a visual pop-up.
#3	Google Hangout	Low	Application alerts the driver for a new message with an audio beep. Driver can click and respond to the message.
#4	Google Hangout (Voice only)	Low	Same as an interaction #3, but all interactions are voice-based.
#5	GeoTask	Low	Application sends an audio ding and pop-up showing the reminder that the driver made when he passes by a specific location.
#6	Speed Cameras	High	Application alerts the driver with an audio message 'speed limit exceeded' when the driver exceeds the posted speed limit. The screen shows driver's current speed and the speed limit.
#7	Foursquare	Low	Application sends an audio ding and sends a notification to show nearby restaurants when the driver enters the new city. When the user clicks the notification, the list of nearby restaurants is shown.

Table 4.7: Seven interactions that were used in Gremlin's evaluation

instance, a social-media based application alerting the driver about a trending restaurant may become irrelevant if the driver passes by that restaurant. Using `gremlin_cancel_notify`, an application can cancel its `gremlin_begin_notify` call. When it happens, `gremlin_cancel_notify` will return with `success` value of 1 and `gremlin_begin_notify` will return with `success` vale of 0. Both functions will return immediately after the cancel call is made.

4.4 Evaluation

I evaluated Gremlin by comparing its results to a usability expert's opinion. I evaluated 7 different Android applications' interactions that may be used in a vehicular setting. I also calculated system overhead of various component of Gremlin.

4.4.1 Setup

I executed Gremlin on a Nexus 9 tablet running a custom version of Android 5.0.1 that incorporates Gremlin. I have modified the Android system as described in Section 4.3

I selected 7 interactions for the evaluation. They were selected because of their potential usage in a vehicular environment. They are interactions from well-know Android applications (Waze, Google Hangout, etc.). The first two interactions include Waze alerting the driver about a hazardous road condition and the presence of traffic camera. The next two interactions are an interaction from a regular (text-based) Google Hangout (an Internet messaging app) and an interaction from a voice-based Google Hangout (using ReaditToMe app). The rest are GeoTask delivering a location-based reminder, Speed Cameras alerting the driver about exceeding the speed limit, and Foursquare notifying the driver about nearby restaurants. Two interactions (Waze's alert on hazardous road condition and Speed Cameras' alert on exceeding the speed limit) have higher priority than driving. Details of each interaction are described in Table 4.7

Since Gremlin needs to perform an offline analysis to obtain attention demand estimates, I have collected five different traces for each interaction using Gremlin's interaction capture tool. Each trace was collected by simulating driving in various cities, simulating a message conversation, etc. Although the traces were not collected from actual users, it represents a good range of actions that a user would have performed. These traces were analyzed by Gremlin's interaction analyzer. Gremlin's estimated 95% confidence intervals of each interaction's $V_{completion\_time}$ and $C_{completion\_time}$ are shown in Table 4.8. Gremlin utilized the upper bound of the confidence interval when making the decision.

Interaction number	$V_{completion\_time}$	$C_{completion\_time}$
Interaction #1	[1.132s, 1.787s]	[0.066s, 0.066s]
Interaction #2	[1.014s, 1.136s]	[0.004s, 0.004s]
Interaction #3	[13.660s, 18.645s]	[0.004s, 0.004s]
Interaction #4	[0s, 0s]	[0.129s, 0.212s]
Interaction #5	[1.337s, 2.227s]	[0.004s, 0.004s]
Interaction #6	[0.691s, 0.706s]	[0.066s, 0.066s]
Interaction #7	[10.118s, 22.773s]	[0.004s, 0.004s]

Table 4.8: 95% confidence interval for seven interactions' $V_{completion\_time}$ and $C_{completion\_time}$. Demand units are in seconds and represented as [lower bound, upper bound]

I evaluated these 7 interactions for 3 driving scenarios: low, medium, and high attention-demanding driving condition. Details of each driving scenario including estimated $V_{deadline}$ and $C_{deadline}$ are illustrated in Table 4.9. For each driving scenario and interaction combination, Gremlin makes a decision on whether to allow an interaction or defer it to later.

4.4.2 Comparison with usability expert

I evaluated the quality of Gremlin's result by comparing them with a usability expert's decision. Prior to receiving expert's opinion, I informed the expert about the details of an interaction and the driving scenarios. I asked the usability expert to share his thoughts on whether allowing each interaction at a certain driving scenario would be desired or not be desired. I did not share Gremlin's result prior to expert's assessment.

Table 4.10 compares the results of Gremlin and expert's opinion. For each form of testing, a check indicates allowing the interaction and an x indicates not delivering the interaction. Gremlin's decision matched with expert's decision on 90% of the interaction/driving scenario combinations.

Gremlin had one false positive (allowing an interaction when it is not supposed to). It occurred for Waze's alert about the traffic camera during the medium demanding

Driving scenario	Description	$V_{deadline}$	$C_{deadline}$
Low attention demanding	Driving at a low speed on a straight rural road. Experienced driver. Low traffic volume. (speed=20mph, lane width=12 feet, curve radius=0 meters)	3.243s	0.274s
Medium attention demanding	Going through a easy curve at a moderate speed on a shorter lane width road. Experienced driver. High traffic volume. (speed=35mph, lane width=11 feet, curve radius=592 meters (2 degree of curve))	2.229s	0.152s
High attention demanding	Going through a sharp curve at a high speed on a highway. Novice driver. Low traffic volume. (speed=50mph, lane width=12 feet, curve radius=60 meters (30 degree of curve))	0.615s	0.073s

Table 4.9: Three driving scenarios that were used in Gremlin's evaluation and their $V_{deadline}$ and $C_{deadline}$ values

driving scenario. The expert did not allow this interaction, because it forces the driver to look at the screen to understand the content of the interaction as the audio output (a ding) does not contain any useful information. Gremlin allowed this interaction, because it believed that the driver can understand the content without negatively impacting driving.

There was also one false negative. Gremlin did not allow GeoTask's interaction when the driver was in the high-demand scenario. The expert felt that the GeoTask interaction should be allowed in all scenarios, including the high-demand scenario in which Gremlin disallowed it, because the driver expects this output to occur at the given location. Since the driver has previously set this reminder, the activation of the reminder should not come as a surprise. Gremlin lacks the context to make this type of inference about the level of surprise in different interactions.

	Low demand		Medium demand		High demand	
Interactions	Gremlin	Expert	Gremlin	Expert	Gremlin	Expert
Interaction #1	✓	✓	✓	✓	✓	✓
Interaction #2	✓	✓	✓	×	×	×
Interaction #3	×	×	×	×	×	×
Interaction #4	✓	✓	×	×	×	×
Interaction #5	✓	✓	✓	✓	×	✓
Interaction #6	✓	✓	✓	✓	✓	✓
Interaction #7	×	×	×	×	×	×

This table compares Gremlin's results against those produced by an usability expert. Columns represent three driving scenarios from low attention demanding, medium attention demanding, and high attention demanding situation. For each combination of interaction and driving scenario, the table shows Gremlin's result and the expert's opinion. A × indicates not allowing the interaction and a ✓ indicates allowing the interaction. Cases where Gremlin and the expert disagreed are highlighted in gray.

Table 4.10: Comparison of Gremlin's decision mechanism with usability expert's opinion

4.4.2.1 Assessment of Gremlin

The comparison of Gremlin's result with expert's opinion illustrates the promising value of Gremlin. However, the expert illustrated that his decision was based on his opinion and it may differ from one expert to another. Gremlin's decision was compared to one expert's opinion. Although the expert agreed with Gremlin's approach of managing driver's attention, the expert illustrated that in order for Gremlin to be fully utilized in a real-world, Gremlin's model needs to consider various factors such as driver's familiarity with road, street conditions, weather, etc.

The key contribution of Gremlin is that it matches the expert's recommendation much better than the existing solutions. Table 4.11 illustrates how two existing solutions match the expert's recommendation. The first evaluated solution is a policy that only allows text-based interactions while a vehicle is stopped and always allows all voice-based interactions. This is a typical policy implemented in current in-vehicle infotainment systems such as Chevrolet MYLINK [41], Toyota Entune [110], and Mazda Connect [73]. This policy is unclear with respect to non-text visual interac-

	Low demand		Medium demand		High demand	
Interactions	Current vehicle policy	Expert	Current vehicle policy	Expert	Current vehicle policy	Expert
Interaction #1	×	✓	×	✓	×	✓
Interaction #2	×	✓	×	×	×	×
Interaction #3	×	×	×	×	×	×
Interaction #4	✓	✓	✓	×	✓	×
Interaction #5	×	✓	×	✓	×	✓
Interaction #6	×	✓	×	✓	×	✓
Interaction #7	×	×	×	×	×	×
Interactions	Do not disturb policy	Expert	Do not disturb policy	Expert	Do not disturb policy	Expert
Interaction #1	✓	✓	×	✓	×	✓
Interaction #2	✓	✓	×	×	×	×
Interaction #3	✓	×	×	×	×	×
Interaction #4	✓	✓	×	×	×	×
Interaction #5	✓	✓	×	✓	×	✓
Interaction #6	✓	✓	×	✓	×	✓
Interaction #7	✓	×	×	×	×	×

This table compares the results of two existing solutions against those produced by an usability expert. The first is a typical scheduling policy of current in-vehicle infotainment systems. A visual text-based interaction is not allowed while the vehicle is moving. Meanwhile, a voice-based interaction is allowed at all times. The second is a 'do not disturb' policy that only allows interactions during the low demand and does not allow any interactions during more attention demanding scenarios.

Table 4.11: Comparison of other policies with usability expert's opinion

tions. However, none of the tested interactions fall into that category (i.e., all tested visual interactions contained text). The second existing solution is a policy that denies all interactions in medium and high attention demanding driving scenarios and allows all interactions in a low attention demanding situation. The first existing solution matches with expert's opinion in 9 cases out of 21 (43% accuracy). The second solution matches 13 out of 21 (62% accuracy). Both approaches have significantly lower accuracy than Gremlin. With the promising result, Gremlin illustrates the first step of managing driver's attention and delivering the interaction only when the driver has enough attention.

4.4.3 Microbenchmarks

First, the overhead of running the interaction capture tool was measured. I recorded the GUI response time, the time between the user's click event and the application's complete GUI rendering. The higher GUI response time is, it takes longer for a user to see the result of her click event. I measured the overhead in two applications: Google Hangout and Waze. Google Hangout was chosen because it represents a highly interactive yet graphically not heavy application. Waze was chosen to represent less interactive but graphically heavy application. For each application, I recorded touch events of the user interacting with an application. I replayed the same touch events 10 times using RERAN [42] for two conditions: baseline and when the capture tool is actively capturing the interaction. The GUI response time was calculated for each touch event and each scenario's average GUI response time is reported in Table 4.12. Gremlin's interaction capture tool only imposed additional 5ms to the GUI response time for Google Hangout application, and 26ms overhead to graphic-heavy Waze application. Although 26ms overhead appears to be large, it is still small enough to be considered within human's non-perceivable range. The higher overhead for graphic heavy application is caused by the increase size in captured video. As more portions of the screen gets changed constantly, video encoder needs to create more frames and cannot use the older frames.

Since the capture tool is only actively capturing the inputs and the outputs while a driver is interacting with an interaction, this small overhead will only persist only in short duration. Most of application's interactions become completed within a minute. After the interaction is finished, the capture tool does not perform any active capturing and waits in the background for the next interaction to capture.

I have also measured the storage overhead of Gremlin's capture tool for all 5 applications that was used for the evaluation. Although it varied from an application to another, Gremlin's interaction capture tool generated about 590kB of data per

Application	Baseline	During the capture	Overhead
Google Hangout	19.549ms	24.368ms	4.819ms
Waze	121.701ms	147.696ms	25.995ms

Table 4.12: Overhead of Gremlin's interaction capture tool, measured by the GUI response time (ms) for two applications

second on average to capture both video and GUI dump information. When capturing the audio, Gremlin imposes the storage overhead of 426kB per second for the audio output and 37kB per second for the audio input. Capturing the audio output has a higher storage overhead, since Gremlin captures the raw audio data instead of encoded audio data as it does with the audio input. Considering the modern mobile devices having gigabytes of storage, I conclude that the mobile device can handle this storage overhead. Also, stored data can be removed from the device, once it gets transferred to the offline analyzer. The transfer can occur concurrently over the cellular network to minimize the storage overhead. Therefore, the storage overhead is only temporary.

Furthermore, I have analyzed the performance of Gremlin's offline analyzer. The performance of Gremlin's interaction analyzer is less significant since it performs the analysis offline. Nonetheless, interaction analyzer analyzed an interaction in about 12.7 seconds on average on a server using 4 cores of a 3.1Ghz Xeon E5-2687W processor with 16GB of ram. The analysis time was dependent on the complexity of screen for video processing and the length of recorded interaction. For one second of recorded interaction, the analyzer took about 1.08 seconds to perform visual demand analysis. To analyze one audio frame that contains speech, it took analyzer about 1.6 seconds to analyze the frame. Table 4.13 illustrates average performance of Gremlin's offline analyzer for analyzing five of the same interactions.

The overhead of GUI response time during Gremlin's scheduling phase was also measured. However, the overhead was very minimal almost non-distinguishable from the baseline case. Since Gremlin does not use any disk I/O and only periodically

Interaction number	# of video frames	# of audio recordings	Processing time
Interaction #1	13.2	1	15.6s
Interaction #2	24.6	1	27.2s
Interaction #3	20.2	1	16.2s
Interaction #4	0	8	12.6s
Interaction #5	2.8	1	4.8s
Interaction #6	2.8	1.4	4.6s
Interaction #7	7.6	1	8.4s

Table 4.13: Average number of video frames and audio recordings that Gremlin analyzed for each interaction and its average processing time in seconds.

computes the new attention supply during the scheduling phase, I did not expect any significant overhead.

4.5 Limitations

Gremlin's initial results are promising. I have shown that Gremlin can manage driver's attention and deliver an interaction only when the driver can handle it. However, for Gremlin to be deployed in a real world, there are many challenges to overcome.

First of all, the attention model needs to be improved. Although Gremlin captures the important factors that affect driver's available attention, it is not complete. Adding factors, such as traffic condition, driver's age, time of day, fatigue, road familiarity, etc, would strengthen the attention supply model. To improve the attention demand model, Gremlin could incorporate driver's familiarity with an interaction, expectancy of an interaction, etc.

Additionally, Gremlin can improve its model on estimating the priority level of an interaction. In the current implementation, Gremlin accepts the priority level that is defined by the application. Although the application is in the best position to assess the priority level of interactions that it initiates, Gremlin can not blindly rely on the application's ranking of priority level. Some applications may purposely overestimate their priority level to game the system. Furthermore, the interaction's priority level may differ from one person to another. Some people are social networking addicts and others care not a whit for the latest updates from their friends. Therefore, Gremlin needs to have a mechanism to adjust priority level of an interaction based on driver's feedback and past behavior (e.g., did the user always dismissed this interaction right away?).

Gremlin's core decision making mechanism appears to be reasonable. With a strengthened attention and priority level model, Gremlin can be deployed in a real vehicle helping drivers from getting distracted by attention-demanding interactions.

Despite Gremlin's limitations, Gremlin's promising result illustrates that driver-attention aware scheduling of an interaction can improve the usability of an application since its interactions are only allowed when the driver has enough spare attention.

CHAPTER V

Related work

5.1 Attaining Low-Latency Continuous Interactivity over the Wide-Area with Speculation

Speculative execution is a general technique for reducing perceived latency. In the domain of distributed systems, Crom [75] allows Web browsers to speculatively execute javascript event handlers (which can prefetch server content, for example) in temporary shadow contexts of the browser. Mosh provides a more responsive remote terminal experience by permitting the client to speculate on the terminal screen's future content, by leveraging the observation that most keystrokes for a terminal application are echoed verbatim to the screen [118]. The authors of [65] show that by speculating on remote desktop and VNC server responses, clients can achieve lower perceived response latency, albeit at the cost of occasional visual artifacts on misprediction. A common theme of this prior work is to build core speculation (e.g. state prediction, state generation) into the client. In contrast, Outatime performs speculation at the server. This is because client vs. server graphical rendering capabilities can differ by orders of magnitude, and clients cannot be reliably counted on to render (regular or speculative) frames.

Time Warp [59] improved distributed simulation performance by speculatively executing computation on distributed nodes. However, to support speculation, it made many assumptions that would be inappropriate for game development, e.g., processes cannot use heap storage.

Outatime is a specific instance of application-specific speculation, defined by Wester et al. [116] and applied to the Speculator system. According to the taxonomy of Wester et al., Outatime implements novel application-specific policies for creating speculations, speculative output, and rollback.

Alternatives to cloud streaming such as HTML5 are intriguing, though interactive games have had stronger performance demands than what browsers tend to offer. Even with native client execution [28, 43], the benefits of cloud-hosting and Outatime (instant start, easier management, etc.) still apply.

Outatime's use of IBR to compensate for navigation events is inspired by work in the graphics community. Early efforts focused on reducing (non-network) latency of virtual reality simulations using basic movement prediction and primitive image warping [94]. The authors of [108] apply hardware-supported IBR to reduce local scene rendering latency. The authors of [71] investigated the use of IBR for network latency compensation of thin clients. IBR alone is well-suited for rendering novel views of static scenes (*e.g.,* museums) but fails to cope with dynamic scenes (*e.g.,* moving entities) with heavy user interaction (impulse events), as is standard in games. In contrast, Outatime's speculative execution handles dynamic scenes and user interaction.

5.2 Verifying User Interface Properties for Vehicular Applications

To the best of my knowledge, AMC is the first tool to apply model checking to verify user interface properties for environments such as vehicles.

Model checking has been used to solve a diverse set of problems such as finding software bugs [18, 19, 120] and validating protocols [15, 112]. When model checking has been applied to testing application GUIs, it has been used both to discover the screens that a user would encounter and to check for faults such as GUI behavior that does not match an application specification.

Most GUI model checking systems require a pre-specified model of the GUI states and transitions [30, 85, 103]. For instance, the work of Takala et al. requires detailed definitions of GUI states and the locations of buttons within each state. Providing such a detailed model is an onerous task that requires substantial knowledge of the application's behavior.

Salvucci [92] created a cognitive model of using in-vehicle interfaces that predicts the time required by a driver to perform interactions with such interfaces. Such a model could be used to refine the heuristics (e.g., for maximum task length) that AMC uses to check for violations.

Several systems have attempted to reverse-engineer the GUI model using techniques similar to those employed by AMC. One such method is to transparently record a user's interactions with the application and generate a model based on those observations [17, 46]. The inferred model must be subsequently validated by the user. Thus, unlike AMC, which explores the GUI autonomously, these methods require a substantial amount of user time to generate the state model.

Similar to AMC's state exploration, a few methods do not require manual interaction to infer GUI states. GUI Ripping [74] infers the GUI model by automatically exploring the application in a depth-first manner. GUI Ripping defines the state by extracting all widgets, properties, and values of the window. Amalfitano et al. [9] apply a similar technique to Android applications, but their system requires application source code to detect all buttons in a state. AutoBlackTest [70] incrementally builds a GUI model as it explores by employing heuristics about what a state should be. None of these approaches deal well with applications that have both mobile and cloud components. It is challenging to checkpoint such applications, and they can exhibit the seemingly non-deterministic behavior discussed in Section 3.2.3.

AMC needs no external information to detect violations since the design principles it verifies are not application-specific. In contrast, most of the cited prior systems require some specification of the correct behavior of the program; for instance that a particular transition should cause the GUI to display a given state. Alternatively, systems such as AutoBlackTest generate test cases automatically, but leave it to the tester to manually verify the output of each test case. Thus, substantial manual effort is required either before or after exploration. AMC avoids this manual effort except in rare cases where it cannot make a definitive judgment.

There has been substantial effort in the usability community to determine best practices for vehicular interface design. This effort has resulted in quantitative best practice guidelines [29, 45, 77, 97, 104]. Currently, industry HMI experts use these and similar guidelines to manually verify the user interfaces of vehicular applications. AMC applies the same guidelines, but does so in an automated fashion. AMC is the first automated tool to validate vehicular interfaces.

5.3 Scheduling Application-Initiated Interactions in Vehicular Computing

In the past, the focus of avoiding poorly-time notification has been in a desktop environment. There have been many studies that measure the detrimental effect of poorly-timed notifications in a desktop environment [11, 26, 56]. Determining the best time to interrupt a user for a notification has been studied extensively [6, 39, 50, 51, 55]. Horvitz et al. [51] developed PRIORITIES, an desktop e-mail notification system that uses a Bayesian model to infer the user's available attention level and compute the expected cost of interruption and deferring alerts. When the benefit of an alert outweighs the cost of interruption, the system delivers the notification to the user. I agree with the principles of this work, but argue that such solutions should be implemented by the OS, as in Gremlin, rather than by individual applications.

Recently, there has been research on determining proper task break points for mobile devices. Fischer et al. [36] determined the end of mobile device interactions to deliver notifications at such instances. Okoshi et al. [82] determined accurate application-specific break points, during which the user can be interrupted while she is using an application. Ho et al. [49] determined when the user is transitioning from one physical activity to another (e.g., from sitting to walking) using body-worn accelerometers and used those moments to deliver notifications. These prior systems do not consider the importance of the notification, nor do they consider the possibility of interrupting an activity when user has spare attention to handle a new secondary task. Gremlin aims to initiate appropriate interactions even when doing so requires the user to interrupt a current task.

Kern et al. [61] proposed a notification design that senses the user's environment and delivers socially acceptable notification modality to the user. It can sense when a user is in a lecture and knows not to disrupt the user with a loud noise. This design is similar to Gremlin, but in a very different environment. It detects four environments and six user activities. Furthermore, it treats all notifications with the same importance.

In vehicular computing, Green has illustrated the need for a workload manager, which regulates the flow of information to drivers that could interfere with driving [44]. Green suggested that a workload manager should divert any interactions that distracts the driver (e.g., an incoming phone call to a voice-mail when a driver is turning an intersection). Although Green has suggested that many vehicular companies are researching to build such workload manager, currently available in-vehicle systems, such as MYLINK [41] and Toyota Entue [110], only employ a simple workload manager. It performs pre-determined coarse grained scheduling policy that ignores changes in driver's available attention in regards to varying driving conditions and different amounts of attention demanded by each interaction as Gremlin would do.

The closest system to Gremlin is the work done by Kim et al. [62]. Kim et al. used body-worn sensor and vehicular sensor data to determine the opportune time to interrupt the user without disrupting her driving ability. The study utilizes on driver's past behavior with an assumption that driver has the best knowledge when to interrupt himself or not. Unfortunately, this approach does not consider varying attention demands that can be imposed by different interactions. The study only looks at interruptibility and how long a user can be interrupted for. The study also does not consider the priority level of an interaction. Other recent studies have focused on measuring changes in driver's workload based on measurable factors, such as pupil dilation [91], heart-rate variability [86], and driver movement changes on a seat [16]. Yet, they have not attempted to identify and utilize driver's available attention for secondary tasks as Gremlin does.

Additionally, in contrast to all these prior approaches, Gremlin provides a specific framework for cooperation between applications and the OS to determine when to initiate new interactions. Thus, a major focus of Gremlin is to determine how best to manage attention as a service provided by the mobile operating system.

CHAPTER VI

Conclusion

In this dissertation, I have created system support that enables users to interact smoothly with mobile applications when wireless network connectivity is poor and when the user's attention is limited.

To remedy poor application usability caused by high mobile network latency, I have created a novel cloud gaming prototype, Outatime. I have demonstrated Outatime on Doom 3, a twitch-based first person shooter, and Fable 3, an action role-playing game because they belong to popular game genres that demand fast response time. Outatime can provide smooth usability of cloud-based gaming application even with the network latency as high as 100ms. This leads me to be optimistic about Outatime's applicability across other game genres.

To tackle the problem of poor application usability caused by attention conflict between user's primary activity and mobile applications, I have focused on improving application usability in vehicular computing environment. I have created a toolkit, AMC, that assists developers to determine where best-practice vehicular UI guideline violations occur in their applications. Furthermore, to minimize distraction caused by application-initiated interactions, I have developed Gremlin, which schedules such interactions without overloading driver's available attention. Both AMC and Gremlin focuses on minimizing distractions caused by an application, which in turn leads to better usability of an application.

By using AMC, application developers can get much needed help in creating less disrupting UI. Main technique behind AMC is based on model checking that allows automatic exploration of an application's interface, verifying that particular elements of that interface meet the appropriate guidelines. This provides feedback in minutes or hours, and identifies most violations without false positives. The few remaining areas of concern can be the focus of human expert review or can be used to target the designer's efforts more efficiently, simplifying the design and verification process.

With Gremlin, interactions initiated by vehicular applications can receive sufficient user attention to be fully usable. Gremlin continuously estimates the driver's available visual and cognitive attention and predicts the attention demand of an interaction. By making scheduling decisions based on the attention supply and demand, Gremlin ensures that an interaction does not become a distraction by demanding too much of driver's limited attention. Gremlin's decisions follow very closely to a expert's decisions. Such promising result shows that Gremlin's design principles can be extended to other attention-limited environments such as (e.g., walking, working, in a meeting) in the future.

I envision that treating user attention as a resource is going to be critical aspect of human-mobile application interaction in the future as people use mobile applications in an auxiliary fashion. I also envision that more mobile applications will be dependent on their remote servers to function, and dealing with poor mobile network will be crucial. In this dissertation, I have laid out toolkits and designs that can be used by applications to improve usability degradation caused by poor wireless network and user's limited available attention.

APPENDIX

APPENDIX

Computing odds ratio

To obtain the odds ratio for a secondary activity, the study [121] analyzes the dataset to determine the cause of CNC. The study analyzes 5 seconds before a CNC and the 1 second after a CNC to determine which secondary activity is responsible for the CNC. For instance, if the driver was changing the radio station 3 seconds before a CNC, the study will mark that the changing the radio station as the cause of this CNC. After analyzing each CNC data and determining the causing secondary activity, the study counts the total number of CNC cases caused by the particular secondary activity (e.g., reading, changing CD). Then, the study counts the total number of CNC cases that are caused with the absence of driver using the secondary activity.

With these two numbers, the study first computes the 'exposure odds of a secondary task in CNC' by dividing the number of CNC cases caused by the secondary activity by the total number of CNC cases that are not caused by the secondary activity. For example, there are 815 cases of CNC that the study analyzed. Among this 815 cases, there were 10 cases of CNC caused by a reading activity. The calculated exposure odds for reading activity in CNC is 0.012 (10/805).

To compute the odds ratio, the exposure odds in CNC gets divided by the exposure odds in baseline cases (where there are no crashes or near crashes). This is standard statistic formula used when computing the odds ratio of an event. The baseline exposure odds is computed by analyzing traces of the same drivers (who were part of CNC cases), but on a different date but in similar time, road location, similar day of week, and did not get involved in a crash or near crash. The study found 10008 baseline cases that did not involve any CNC with the same drivers. The study analyzes each baseline cases to determine if a secondary task was present in the same 6 second window. Going back to the reading activity example, there are 92 cases where the driver was reading but did not get involved in a CNC and 9916 cases where the driver was not reading at all (and not involved in a CNC). Therefore, the exposure odds of reading activity in the baseline case is 0.009. By diving exposure odds in CNC by the exposure odds in the baseline, the odds ratio of reading while driving is computed as 1.33 (0.012/0.009).

To my knowledge, SAE study is the most up-to-date study that overcomes the shortcomings of previous studies. Previous studies [64] have used random baseline (instead of matching the same drivers in CNC cases), had errors in determining the cause of CNC, only analyzed at fault-accidents. Therefore, Gremlin utilizes this study to determine the maximum allowed cognitive threshold.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Amazon appstream. `http://aws.amazon.com/appstream`.

[2] Nvidia grid cloud gaming. `http://shield.nvidia.com/grid`.

[3] Sony playstation now streaming. `http://us.playstation.com/playstationnow`.

[4] Sponging is no longer a myth. `http://youtu.be/Bt433RepDwM`.

[5] A technical companion to windows embedded automotive 7. Technical report, Microsoft, July 2010.

[6] Piotr D. Adamczyk and Brian P. Bailey. If not now, when?: The effects of interruption at different moments within task execution. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 271–278, Vienna, Austria, April 2004.

[7] Mark Allman. Comments on bufferbloat. *SIGCOMM Comput. Commun. Rev.*, 43(1):30–37, January 2012.

[8] D. Alan Allport, Barbara Antonis, and Patricia Reynolds. On the division of attention: A disproof of the single channel hypothesis. *Quarterly Journal of Experimental Psychology*, 24(2):225–235, 1972.

[9] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI crawling-based technique for android mobile application testing. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261, Berlin, Germany, March 2011.

[10] Smartphone use behind the wheel survey. `http://about.att.com/content/dam/snrdocs/2015%20It%20Can%20Wait%20Report_Smartphone%20Use%20Behind%20the%20Wheel%20.pdf`.

[11] Brian P. Bailey and Joseph A. Konstan. On the need for attention-aware systems: Measuring effects of interruption on task performance, error rate, and affective state. *Computers in Human Behavior*, 22(4):685–708, 2006.

[12] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *NetGames'04*, pages 144–151, New York, NY, USA, 2004. ACM.

[13] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM'08*, pages 389–400, New York, NY, USA, 2008. ACM.

[14] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI'06*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[15] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[16] Andreas Braun, Sebastian Frank, Martin Majewski, and Xiaofeng Wang. Capseat: Capacitive proximity sensing for automotive activity recognition. In *Proceedings of the 7th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, pages 225–232, Nottingham, United Kingdom, 2015.

[17] Penelope A. Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *Proceedings of the 23rd IEEE/ACM international conference on Automated software engineering*, pages 333–342, Atlanta, GA, November 2007.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 209–224, December 2008.

[19] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[20] Ben Caldwell, Michael Cooper, Loretta Guarino Reid, and Gregg Vanderheiden. Web Content Accessibility Guidelines (WCAG) 2.0, 2008. `http://www.w3.org/TR/WCAG20/#contrast-ratiodef`.

[21] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. How sensitive are online gamers to network quality? *Commun. ACM*, 49(11):34–38, November 2006.

[22] Mark Claypool, David Finkel, Alexander Grant, and Michael Solano. Thin to win?: network performance analysis of the onlive thin client game system. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, NetGames '12, pages 1:1–1:6, Piscataway, NJ, USA, 2012. IEEE Press.

[23] CMU Sphinx, `http://cmusphinx.sourceforge.net/`. *PocketSphinx*.

[24] Catherine Courage, Paul. Milgram, and Alison Smiley. An investigation of attentional demand in a simulated driving environment. In *Proceedings of the*

Human Factors and Ergonomics Society Annual Meeting, San Diego, CA, July 2000.

[25] Eduardo Cuervo. *Enhancing Mobile Devices through Code Offload*. PhD thesis, Duke University, 2012.

[26] Mary Czerwinski, Eric Horvitz, and Susan Wilhite. A diary study of task switching and interruptions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 175–182, Vienna, Austria, April 2004.

[27] Matthias Dick, Oliver Wellnitz, and Lars Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *NetGames'05*, pages 1–7, New York, NY, USA, 2005. ACM.

[28] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 339–354, Berkeley, CA, USA, 2008. USENIX Association.

[29] Driver Focus-Telematics Working Group. Statement of principles, criteria and verification procedures on driver interactions with advanced in-vehicle information and communication systems. Technical report, Alliance of Automobile Manufacturers, June 2003.

[30] Matthew B. Dwyer, Vicki Carr, and Laura Hines. Model checking graphical user interfaces using abstractions. In *Proceedings of the 6th European Software Engineering Conference*, pages 244–261, Zurich, Switzerland, September 1997.

[31] Engadget. Microsoft's delorean is a cloud gaming system that knows what you'll do next. `http://www.engadget.com/2014/08/23/microsoft-delorean/`, 8 2014.

[32] Johan Engstrom, Emma Johansson, and Joakim Ostlund. Effects of visual and cognitive load in real and simulated motorway driving. *Transportation Research Part F: Traffic Psychology and Behaviour*, 8(2):97–120, 2005.

[33] Epic Games. Unreal graphics programming. `https://docs.unrealengine.com/latest/INT/Programming/Rendering/index.html`.

[34] Epic Games. Unreal networking architecture. `http://udn.epicgames.com/Three/NetworkingOverview.html`.

[35] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2007.

[36] Joel E. Fischer, Chris Greenhalgh, and Steve Benford. Investigating episodes of mobile phone activity as indicators of opportune moments to deliver notifications. In *Proceedings of the 13th International Conference on Human Computer*

Interaction with Mobile Devices and Services, pages 181–190, Stockholm, Sweden, August 2011.

[37] Paul M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47:381–391, 1954.

[38] Flurry. Apps solidify leadership six years into the mobile revolution. `http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution`, 4 2014.

[39] James Fogarty, Scott E. Hudson, Christopher G. Atkeson, Daniel Avrahami, Jodi Forlizzi, Sara Kiesler, Johnny C. Lee, and Jie Yang. Predicting human interruptibility with sensors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 12(1):119–146, March 2005.

[40] Gamespot. Microsoft researching cloud gaming solution that hides latency by predicting your actions. `http://www.gamespot.com/articles/microsoft-researching-cloud-gaming-solution-that-h/1100-6421896/`, 8 2014.

[41] General Motors LLC. 2015 Chevrolet MyLink details book. `https://my.gm.com/`.

[42] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 72–81, San Francisco, CA, May 2013.

[43] Google. Native client. `http://youtu.be/Bt433RepDwM`.

[44] Paul Green. Driver distraction, telematics design, and workload managers: Safety issues and solutions. In *SAE Convergence*. Society of Automotive Engineers, 2004.

[45] Paul Green, William Levison, Gretchen Paelke, and Colleen Serafin. Suggested human factors design guidelines for driver information systems. Technical report, The University of Michigan Transportation Research Institute (UMTRI), August 1994.

[46] A.M.P. Grilo, A.C.R. Paiva, and J.P. Faria. Reverse engineering of GUI models for testing. In *Proceedings of the 5th Iberian Conference on Information Systems and Technologies*, pages 1–6, Santiago de Compostela, Spain, June 2010.

[47] Lisbeth Harms and Christopher Patten. Peripheral detection as a measure of driver distraction. a study of memory-based versus system-based navigation in a built-up area. *Transportation Research Part F: Traffic Psychology and Behaviour*, 6(1):23–36, 2003.

[48] Hierarchy Viewer. `http://developer.android.com/tools/help/hierarchy-viewer.html`.

[49] Joyce Ho and Stephen S. Intille. Using context-aware computing to reduce the perceived burden of interruptions from mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 909–918, Portland, Oregon, April 2005.

[50] Eric Horvitz and Johnson Apacible. Learning and reasoning about interruption. In *Proceedings of the 5th International Conference on Multimodal Interfaces*, ICMI '03, pages 20–27, Vancouver, Canada, November 2003.

[51] Eric Horvitz, Andy Jacobs, and David Hovel. Attention-sensitive alerting. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, pages 305–313, Stockholm, Sweden, July 1999.

[52] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *MobiSys'12*, pages 225–238, New York, NY, USA, 2012. ACM.

[53] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. An in-depth study of lte: effect of network protocol and application behavior on performance. In *SIGCOMM'13*, pages 363–374, New York, NY, USA, 2013. ACM.

[54] Intel. QuickSync Programmable Video Processor. `http://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html`.

[55] Shamsi T. Iqbal and Brian P. Bailey. Understanding and developing models for detecting and differentiating breakpoints during interactive tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 697–706, San Jose, California, April 2007.

[56] Shamsi T. Iqbal and Eric Horvitz. Disruption and recovery of computing tasks: Field study, analysis, and directions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 677–686, San Jose, California, April 2007.

[57] Georg Jahn, Astrid Oehme, Josef F. Krems, and Christhard Gelau. Peripheral detection as a workload measure in driving: Effects of traffic complexity and route guidance system use in a driving study. *Transportation Research Part F: Traffic Psychology and Behaviour*, 8(3):255–275, 2005.

[58] A. Hamish Jamson and Natasha Merat. Surrogate in-vehicle information systems and driver behaviour: Effects of visual and cognitive load in simulated rural driving. *Transportation Research Part F: Traffic Psychology and Behaviour*, 8(2):79–96, 2005.

[59] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P.Hontalas, P. Laroche, K. Sturdevant, J. Tupman, Van Warren, J. Weidel, H. Younger, and S. Bellenot. Time Warp operating system. In *SOSP'87*, pages 77–93, Austin, TX, November 1987.

[60] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[61] Nickey Kern and Bernt Schiele. Context-aware notification for wearable computing. In *Proceedings of the 7th IEEE International Symposium on Wearable Computers*, pages 223–230, Washington, DC, October 2003.

[62] SeungJun Kim, Jaemin Chun, and Anind K. Dey. Sensors know when to interrupt you in the car: Detecting driver interruptibility through monitoring of peripheral interactions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI)*, pages 487–496, Seoul, Republic of Korea, April 2015.

[63] J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, DTIC Document, Feb 1975.

[64] S. G. Klauer, T. A. Dingus, V. L. Neale, J. Sudweeks, and D.J. Ramseyi. The impact of driver inattention on near-crash/crash risk: An analysis using the 100-car naturalistic driving study data. Technical report, NHTSA, April 2006.

[65] John R. Lange, Peter A. Dinda, and Samuel Rossoff. Experiences with client-based speculative remote display. In *ATC'08*, pages 419–432, Berkeley, CA, USA, 2008. USENIX Association.

[66] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Alec Wolman, and Jason Flinn. Demo: Delorean: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. MobiSys '14, 2014.

[67] Kyungmin Lee, Jason Flinn, T.J. Giuli, Brian Noble, and Christopher Peplin. AMC: Verifying user interface properties for vehicular applications. In *Proceedings of the 11th International Conference on Mobile Systems, Applications and Services*, pages 1–12, Taipei, Taiwan, June 2013.

[68] Chin-Chiuan Lin. Effects of contrast ratio and text color on visual performance with tft-lcd. *International Journal of Industrial Ergonomics*, 31(2):65–72, 2003.

[69] General Motors LLC. 2014 cadillac cue infotainment system. `https://www.cadillac.com/content/dam/Cadillac/Global/master/nscwebsite/en/home/Owners/Manuals_and_Videos/01_images/2014_Cue_Manual.pdf`.

[70] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: a tool for automatic black-box testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1013–1015, Waikiki, Honolulu, HI, May 2011.

[71] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 7–ff., New York, NY, USA, 1997. ACM.

[72] M.H. Martens and W. van Winsum. Measuring distraction: the peripheral detection task. Technical report, National Highway Traffic Safety Administration, Jun 2000.

[73] Mazda Motor Company. MZD Connect. `http://infotainment.mazdahandsfree.com/communication-sms?language=en-RW`.

[74] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th working conference on reverse engineering*, pages 260–270, Victoria, B.C., Canada, November 2003.

[75] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster web browsing using speculative execution. In *NSDI'10*, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[76] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 75–88, Boston, MA, December 2002.

[77] Yukinobu Nakamura. JAMA guideline for in-vehicle display systems. Technical report, Japan Automobile Manufacturers Association, Oct 2008.

[78] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst.*, 24(4):361–392, November 2006.

[79] Nvidia. Video codec sdk. `https://developer.nvidia.com/nvidia-video-codec-sdk`.

[80] Volkswagen of America Inc. My2016 infotainment and car-net: A new generation of connectivity! `https://www.vwwebsource.com/modules/infotainment2016/docs/2016_Infotainment.pdf`.

[81] Alliance of Automobile Manufacturers. Comments received from the alliance of automobile manufacturers. Accessed at www.regulations.gov, Docket NHTSA-2010-0053, Document Number 0104.

[82] Tadashi Okoshi, Hideyuki Tokuda, and Jin Nakazawa. Attelia: Sensing user's attention status on smart phones. In *16th International Conference on Ubiquitous Computing*, pages 139–142, Seattle, Washington, September 2014.

[83] S. Olsson and P.C. Burns. Measuring driver visual distraction with a peripheral detection task. Technical report, National Highway Traffic Safety Administration, Feb 2008.

[84] https://www.openstreetmap.org.

[85] Ana C.R. Paiva, Joao C.P. Faria, Nikolai Tillmann, and Raul A.M. Vidal. A model-to-implementation mapping tool for automated model-based GUI testing. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, pages 450–464, Manchester, United Kingdom, November 2005.

[86] Jung Wook Park, SeungJun Kim, and Anind Dey. Integrated driving aware system in the real-world: Sensing, computing and feedback. In *Proceedings of the 2016 CHI Conference: Extended Abstracts on Human Factors in Computing Systems*, pages 1591–1597, Santa Clara, CA, 2016.

[87] Christopher J.D Patten, Albert Kircher, Joakim Ostlund, and Lena Nilsson. Using mobile telephones: cognitive workload and attention resource allocation. *Accident Analysis and Prevention*, 36(3):341–350, 2004.

[88] Christopher J.D. Patten, Albert Kircher, Joakim Ostlund, Lena Nilsson, and Svenson Ola. Driver experience and cognitive workload in different traffic environments. *Accident Analysis and Prevention*, 38(5):887–894, 2006.

[89] PCWorld. Popcap games ceo: Android still too fragmented. `http://bit.ly/1hQv8Mn`, Mar 2012.

[90] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In Wu chang Feng, editor, *NETGAMES*, pages 152–156. ACM, 2004.

[91] Rahul Rajan, Ted Selker, and Ian Lane. Task load estimation and mediation using psycho-physiological measures. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*, pages 48–59, Sonoma, CA, 2016.

[92] Dario D. Salvucci. Predicting the effects of in-car interfaces on driver behavior using a cognitive architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 120–127, Seattle, Washington, March 2001.

[93] J.W. Senders, A.B. Kristofferson, W.H Levison, C.W. Dietrich, and Ward J.L. The attentional demand of automobile driving. *Highway Research Record*, (195):15–33, 1967.

[94] Richard HY So and Michael J Griffin. Compensating lags in head-coupled displays using head position prediction and image deflection. *Journal of Aircraft*, 29(6):1064–1068, 1992.

[95] Joel Sommers and Paul Barford. Cell vs. wifi: on the performance of metro area mobile connections. In *IMC'12*, pages 301–314, New York, NY, USA, 2012. ACM.

[96] Matt Stanton, Ben Humberston, Brandon Kase, James F. O'Brien, Kayvon Fatahalian, and Adrien Treuille. Self-refining games using player analytics. *ACM Trans. Graph.*, 33(4):73:1–73:9, July 2014.

[97] A Stevens, A Quimby, A Board, T Kersloot, and P Burns. Design guidelines for safety of in-vehicle information systems. Technical report, Transport Research Laboratory, Feb 2002.

[98] David L. Strayer, Joel M. Cooper, Jonna Turrill, James Coleman, Nate MedeirosWard, and Francesco Biondi. Measuring cognitive distraction in the automobile. Technical report, AAA Foundation for Traffic Safety, Jun 2013.

[99] David L. Strayer, Jonna Turrill, James R. Coleman, Emily V. Ortiz, and Joel M. Cooper. Measuring cognitive distraction in the automobile ii: Assessing in-vehicle voice-based interactive technologies. Technical report, AAA Foundation for Traffic Safety, Oct 2014.

[100] Xianghong Sun, Tom Plocher, and Weina Qu. An empirical study on the smallest comfortable button/icon size on touch screen. In *Proceedings of the 2nd International Conference on Usability and Internationalizationi (UI-HCII)*, pages 615–621, Beijing, China, July 2007.

[101] Srikanth Sundaresan, Walter de Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband internet performance: A view from the gateway. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 134–145, New York, NY, USA, 2011. ACM.

[102] Richard Szeliski. *Computer Vision: Algorithms and Applications.* Springer, 2011.

[103] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an android application. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation*, pages 377–386, Berlin, Germany, March 2011.

[104] Task Force HMI. European statement of principles on human machine interface for in-vehicle information and communication systems. Technical report, Commission of the European Communities, December 1998.

[105] TechCrunch. Microsoft research shows off delorean, its tech for building a lag-free cloud gaming service. `http://techcrunch.com/2014/08/22/microsoft-research-shows-off-delorean-its-tech-for-building-a-lag-free-cloud`, 8 2014.

[106] TechHive. Game developers still not sold on android. `http://www.techhive.com/article/2032740/game-developers-still-not-sold-on-android.html`, Apr 2013.

[107] Tesseract-ocr. `http://code.google.com/p/tesseract-ocr/`.

[108] Jay Torborg and James T. Kajiya. Talisman: Commodity realtime 3d graphics for the pc. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 353–363, New York, NY, USA, 1996. ACM.

[109] Jan E.B. Tornros and Anne K. Bolling. Mobile phone useeffects of handheld and handsfree phones on driving performance. *Accident Analysis & Prevention*, 37(5):902–909, 2005.

[110] Toyota Motor Company. Entune system quick reference guide. `http://www.toyota.com/t3Portal/document/om-s/OM16QTQRG/pdf/OM16QTQRG.pdf`.

[111] Omer Tsimhoni, Herbert Yoo, and Paul Green. Effects of visual demand and in-vehicle task complexity on driving and task performance as assessed by visual occlusion. Technical report, The University of Michigan Transportation Research Institute (UMTRI), December 1999.

[112] T. Tsuchiya and A. Schiper. Model checking of consensus algorithm. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 137 –148, Beijing, China, October 2007.

[113] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[114] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[115] An-Hsiang Wang and Ming-Te Chen. Effects of polarity and luminance contrast on visual performance and vdt display quality. *International Journal of Industrial Ergonomics*, 25(4):415–421, 2000.

[116] Benjamin Wester, Peter M. Chen, and Jason Flinn. Operating system support for application-specific speculation. In *Proceedings of the 6th ACM European Conference on Computer Systems*, April 2011.

[117] Christopher D Wickens. Processing resources and attention. *Multiple-task performance*, pages 3–34, 1991.

[118] Keith Winstein and Hari Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, Boston, MA, June 2012.

[119] Wired. As android rises, app makers tumble into google's matrix of pain. `http://www.wired.com/business/2013/08/android-matrix-of-pain/`, Aug 2013.

[120] Junfeng Yang, Can Sar, and Dawson Engler. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 131–146, Seattle, WA, November 2006.

[121] R Young. Revised odds ratio estimates of secondary tasks: A re-analysis of the 100-car naturalistic driving study data. Technical report, SAE Technical Paper, Jan 2015.

[122] Martina Ziefle. Effects of display resolution on visual performance. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40(4):554–568, 1998.

[123] Martina Ziefle. Effects of display resolution on visual performance. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 40(4):554–568, 1998.

[124] Silvia Zuffi, Carla Brambilla, Giordano Beretta, and Paolo Scala. Human computer interaction: Legibility and contrast. In *14th International Conference on Image Analysis and Processing (ICIAP)*, pages 241–246, Modena, Italy, September 2007.