Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming

Kyungmin Lee[†] David Chu[‡] Eduardo Cuervo[‡] Johannes Kopf[‡]
Yury Degtyarev[◦] Sergey Grizan[◇] Alec Wolman[‡] Jason Flinn[†]
[†]University of Michigan [‡]Microsoft Research
[◦]St. Petersburg Polytechnic University [◇]Siberian Federal University

ABSTRACT

Gaming on phones, tablets and laptops is very popular. Cloud gaming — where remote servers perform game execution and rendering on behalf of thin clients that simply send input and display output frames — promises any device the ability to play any game any time. Unfortunately, the reality is that wide-area network latencies are often prohibitive; cellular, Wi-Fi and even wired residential end host round trip times (RTTs) can exceed 100ms, a threshold above which many gamers tend to deem responsiveness unacceptable.

In this paper, we present Outatime, a speculative execution system for mobile cloud gaming that is able to mask up to 120ms of network latency. Outatime renders speculative frames of future possible outcomes, delivering them to the client one entire RTT ahead of time, and recovers quickly from mis-speculations when they occur. Clients perceive little latency. To achieve this, Outatime combines: 1) future state prediction; 2) state approximation with image-based rendering and event time-shifting; 3) fast state checkpoint and rollback; and 4) state compression for bandwidth savings.

To evaluate the Outatime speculation system, we use two high quality, commercially-released games: a twitch-based first person shooter, Doom 3, and an action role playing game, Fable 3. Through user studies and performance benchmarks, we find that players strongly prefer Outatime to traditional thin-client gaming where the network RTT is fully visible, and that Outatime successfully mimics playing across a low-latency network.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Gaming*

Keywords

Cloud gaming; Speculation; Network latency

MobiSys'15, May 18–22, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2742647.2742656.

1. INTRODUCTION

Gaming is the most popular mobile activity, accounting for nearly a third of the time spent on mobile devices [18]. 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 [32, 40, 44] — 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 [1–3, 10].

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 [6,12,33]. A further delay degradation from 150ms to 250ms lowers user engagement by 75% [9].

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 [21, 22, 35]. In fact, even well-established residential wired last mile links tend to suffer from latencies in excess of 100ms when under load [5,37]. Unlike non-interactive video streaming, buffering is not possible for interactive gaming.

Instead, we propose to mitigate wide-area latency via speculative execution. Our system, *Outatime*,[1] delivers real-time gaming interactivity as fast as — and in some cases, even

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

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.

Future State Prediction: Given the user's historical tendencies and recent behavior, we show that some categories of user actions are highly predictable. We develop a Markov-based prediction model that examines recent user input to forecast expected future input. We 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. Our 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, we 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, we 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. We 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, we 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, we 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, we 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. We have also deployed Outatime to the general public on limited occasions and received positive reception [14, 19, 27, 39]. While we have focused our evaluation on desktop and laptop clients experiencing high network latency, our 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.

The remainder of the paper is organized as follows. §2 reviews game architectures and the impact of latency. §3 presents an overview of the Outatime architecture. §4 and §5 detail our two main methods of speculation. §6 and §7 discusses how we save, load and compress state. §8 introduces application to multiplayer. §9 covers the implementation. §10 evaluates Outatime via user study and performance benchmarks. §11 covers related work and §13 discusses implications of the work.

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

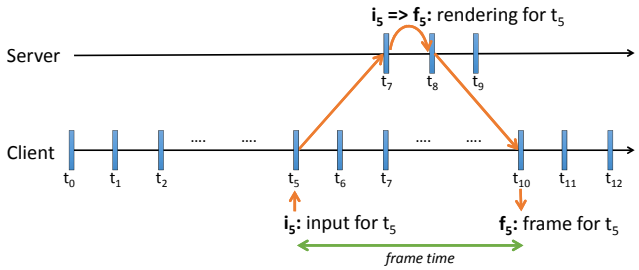
Network latency has an acute effect on interaction for cloud gaming. In standard cloud gaming, the frame time must include the additional overhead of the network RTT, as illustrated in Figure 1a. Let time be discretized into 32ms clock ticks, and let the RTT be 4 ticks (128ms). At t_5, the client reads 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.

3. GOALS AND SYSTEM ARCHITECTURE

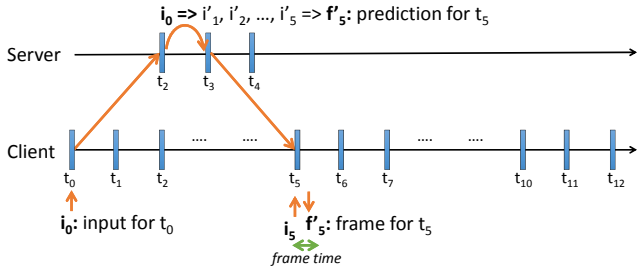
For Outatime, responsiveness is paramount; Outatime's goal is to consistently deliver low frame times (< 32ms) at high frame rate (> 30fps) even in the face of long RTTs. In exchange, we are willing to transmit a higher volume of data and potentially even introduce (small and ephemeral) visual artifacts, 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 1b, the client sends input as

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



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



(b) Outatime: Frame time is negligible.

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

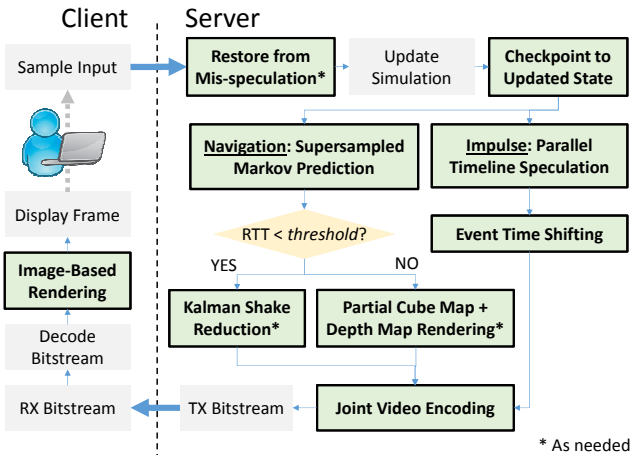


Figure 2: The Outatime Architecture. Bold boxes represent the main areas of this paper's technical focus.

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 $i'_1, i'_2, ..., i'_5$ (we 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 we ensure that the game output is deterministic for a given input [11]. If the input sequence differs, the client approximates the actual state by applying *error compensation* to f'_5 and displays a corrected frame. We describe error compensation in detail in §4.5. 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. 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 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, we 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 we discuss how it is handled in §5.

Figure 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 we show in §10.

4. SPECULATION FOR NAVIGATION

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

4.1 Basic Markov Prediction

We 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. Our 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: $p(N_{t+1}|N_t)$. We predict the most likely navigation vector $\hat{N}_{t+1}$ at the next time step as:

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

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, we predict the most likely value after one RTT as:

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

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

Our 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 we 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 we observed that this assumption of treating the vector components independently does not hurt prediction accuracy. Below in §4.3, we discuss the issue of training in more detail.

4.2 Supersampling

We further refine our 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, we 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, we can often capture at least four samples per tick. We 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 $i \stackrel{+}{=} 0.25$. A summary of navigation prediction accuracy from the user study described in §10 is shown in Figure 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 we show its performance in detail in Figure 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, prediction error below 4° is under the threshold at which output frame differences are perceivable.

Based on these results, we 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, we require additional misprediction compensation mechanisms. Before discussing both of these cases in turn, we first address the question of how much training is needed for successful application of the predictive model.

4.3 Bootstrap Time

Construction of a reasonable Markov Model requires sufficient training data collected during an observation period. Figure 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, we chose to use a single model per coarse-grained skill level (novice or experienced) which is agnostic of the player.

4.4 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 we 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 $+2°, -3°, +3°$. Unfortunately, the user would perceive a shaking effect because the frames would jump by 5° in one direction, and then 6° in another. From our experience with early prototypes, the manifested shakiness was sufficiently noticeable so as to reduce playability.

We apply a Kalman filter [25] 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. For space, we omit technical development of the filter for our problem. One interesting filter modification we make is that we extend the filter to support error accumulation over variable RTT time steps; samples are weighed against an RTT's worth of prediction error.

Using the Kalman formulation, we assume a linear model with Gaussian noise, that is:

$$N_{t+1} = AN_t + \omega \quad (1)$$

$$n_t = N_t + \nu \quad (2)$$

for some state transition matrix A and white Gaussian noise ω. We assign $A = \{a_{ij}\}$ as a matrix which encodes the maximum likelihood Markov transitions where $a_{ij} = 1$ if $i = n_t$ and $j = \hat{N}_{t+\lambda}$, and zero otherwise; $\omega = \mathcal{N}(0, Q_\omega)$ is a normal distribution where Q_ω is the covariance matrix of the dynamic noise; $\nu = \mathcal{N}(0, Q_\nu)$ is a normal distribution where

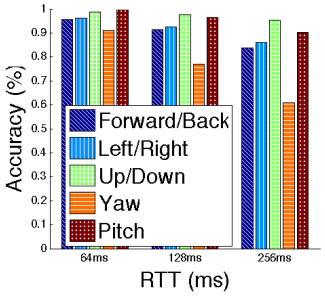
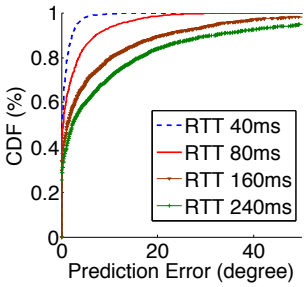
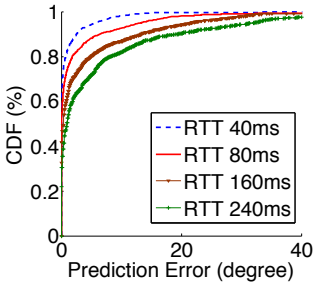


Figure 3: Doom 3 Navigation Prediction Summary. Roll (θ_z) is not an input in Doom 3 and need not be predicted.



(a) Doom 3



(b) Fable 3

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

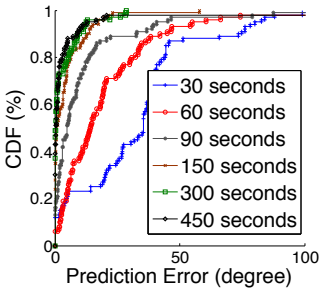


Figure 5: Error Decreases with More Observation Time. Data is for Fable 3 at RTT= 160ms.

Q_ν is the covariance matrix of the sampling noise. We assign covariances Q_ω and Q_ν based on *a priori* observations.

For the base case where the RTT is one clock tick, we generate navigation predictions $\hat{N}_{t+1|t}$ for time $t+1$ given observations up to and including t as follows:

$$\hat{N}_{t+1|t} = A\hat{N}_{t|t} \tag{3}$$

Similarly, navigation error covariance predictions $P_{t+1|t}$ are computed as follows.

$$P_{t+1|t} = AP_{t|t}A^T + Q_\omega \tag{4}$$

When a new measurement n_t arrives, we update the estimate of both the navigation prediction and its error covariance at the current time step as:

$$\hat{N}_{t|t} = \hat{N}_{t|t-1} + G_t(n_t - \hat{N}_{t|t-1}) \tag{5}$$

$$P_{t|t} = P_{t|t-1} - G_t P_{t|t-1} \tag{6}$$

where G_t is the Kalman gain, which is:

$$G_t = P_{t|t-1}[P_{t|t-1} + Q_v]^{-1} \tag{7}$$

Lastly, we initialize the Kalman filter based on the *a priori* distribution: $\hat{N}_{1|0} = E[N_1]$.

Note the feedback loop between the error covariance and Kalman gain weighting favors the measurement n_t when the prediction error is high and favors the prediction $\hat{N}_{t|t-1}$ when error is low. Moreover, the weighting naturally smooths shaking caused by inaccurate prediction by favoring the existing value of $\hat{N}$.

To extend λ timesteps for predictions of $\hat{N}_{t+\lambda|t}$, we compute λ intermediate iterations of Equation (3),(4), (6) and (7). When a new measurement $n_{t+\lambda}$ is received, the accumulation of prediction error $P_{t+\lambda|t}$ will have increased the gain $G_{t+\lambda}$. The large gain then favors the observation when adjusting the estimate in Equation (5).

4.5 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. Our goal in misprediction compensation 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'.

Figure 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.

4.5.1 Image-based Rendering

We 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 we 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 [38]. Figure 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 rendering a normal 2D image by default, we render a cube map [17] centered at the player's position. As shown in Figure 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.

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

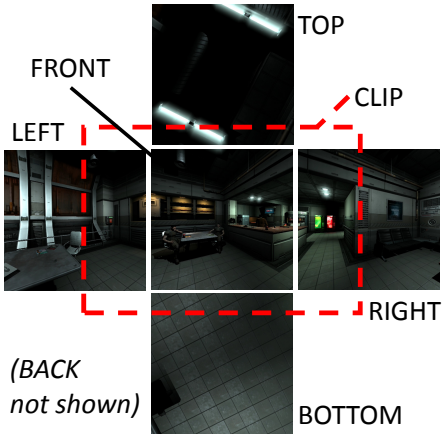


Figure 7: Cube Map Example w/ Doom 3. Clip region shown.

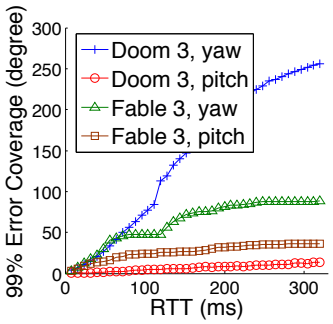
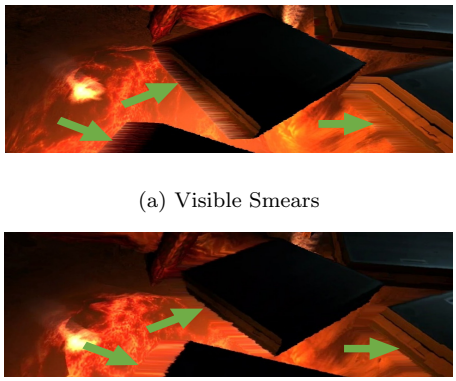


Figure 8: Angular coverage of 99% of prediction errors is much less than 360° even for high RTT.



(a) Visible Smears

(b) Patched Smears

Figure 9: Misprediction's visual artifacts appear as smears which we mitigate.

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. We use the following technique to reduce this overhead.

4.5.2 *Clipped Cube Map*

We 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 for errors in 360° is gratuitous. Therefore, we 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 we render more of the cube. On the other hand, if the prediction variance is low, we render less of the cube. The dotted line in Figure 7 marks the clip region for an example rendering.

In order to size the clip, we 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, we 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 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 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 3) is sufficiently high to obviate consideration of accumulated translation error for the purposes of clipping.

4.5.3 *Patching Visual Artifacts*

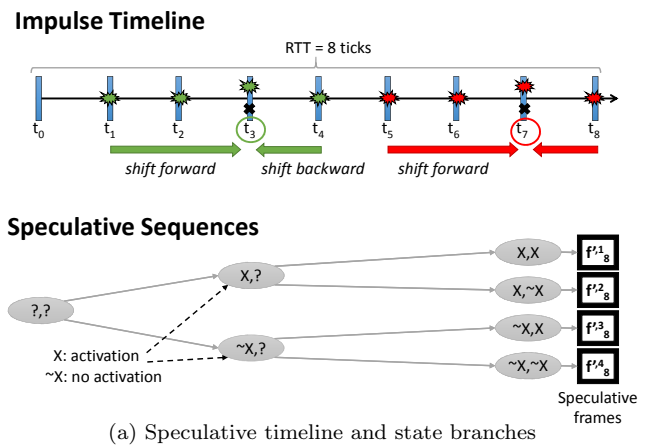
Lastly, we add a technique to mitigate visual artifacts. These occasionally appear when the navigation prediction is wrong and there is significant depth disparity between foreground and background objects. For example, in Figure 9a, the floating stones are much closer to the camera than the lava in the background. As a result, IBR reveals "blind spots" behind the stones as indicated by the green arrows, which are manifest as visual "smears".

Our blind spot patching technique mitigates this problem, as seen in Figure 9b. As part of IBR, we extrude the object borders in the depth map with lightweight image processing which propagates depth values from foreground borders onto the background. However, we 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.

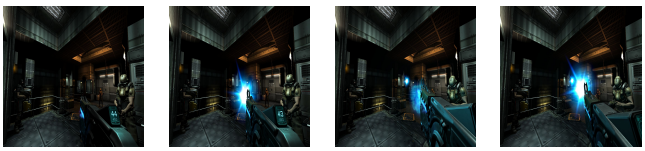
5. SPECULATION FOR IMPULSE EVENTS

The prototypical impulse events are FIRE for first person shooters, and INTERACT (with other characters or objects) for role playing games. We 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 inconsis-



(a) Speculative timeline and state branches



(b) ∼X, ∼X (c) ∼X, X (d) X,∼X (e) X,X

Figure 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.

tencies. 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, we employ a speculation technique for impulse that differs substantially from navigation speculation. Rather than attempt to predict impulse events, instead we 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. We use two state approximations to tame state space explosion: subsampling and time-shifting.

5.1 Subsampling

We 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.

5.2 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 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 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 [6, 33], we 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, we 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, we generate a maximum of four speculative sequences as shown in Figure 10a. When RTT $>$ 256ms, we further lower the subsampling frequency sufficiently to ensure that we 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.

5.3 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 σ = 128ms, Outatime is able to support RTT $\leq$ 128ms for ternary and quaternary impulse events without lowering the subsampling frequency.

5.4 Delay Tolerant Events

We classify any input event that is slow relative to RTT as *delay tolerant*. We use a practical observation to simplify handling of delay tolerant events. According to our 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. Weapon switching takes even longer. Fable 3 delay tolerant events have even higher cool down times. We 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 we 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), we may miss λ frames due to the RTT. During the remaining $\tau - \lambda$ frames, we 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, we 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.

6. FAST CHECKPOINT AND ROLLBACK

As with other systems that perform speculation [30, 42], 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, our continuous 30fps interactivity performance constraints are qualitatively much more demanding, and we highlight how we have managed these requirements.

Unique among speculation systems, we use a hybrid of page-level checkpointing and object-level checkpointing. This is because we 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, we choose either object-level or page-level checkpointing based on the density of SSOs among co-located objects. We 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, we 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 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, we track lifetimes of objects rather than pages for any object the developer has marked for tracking. To rollback a speculation, we delete any object that did not exist at the checkpoint, and restore any objects that were deleted during speculation. Moreover, we 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; we found the savings in checkpoint and rollback time to be very significant in certain cases (§10).

7. 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.

We can achieve a large reduction in bandwidth by observing that frames from different speculations share significant spatial and temporal coherence. Using Figure 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 10b–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, we 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, we 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 [23, 31]. We have reprogrammed these otherwise idle hardware accelerated capabilities for our speculation's joint encoding.

8. MULTIPLAYER

Thus far, we 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, we 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 [8,16]. 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 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, we are interested in extending Outatime's speculative approach in conjunction with AI-led state approximations [7] to better remedy glitches that appear generally in any multiplayer game.

9. IMPLEMENTATION

To prototype Outatime, we 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, we 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. Our experience with Fable 3 was similar

We have made the following key modifications to Doom 3. To enable deterministic speculation, we made changes according to [11] 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, we 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 our *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, we 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.

We 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 [31]. 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. We 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, we also examined the source code for Unreal Engine [16], 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` [15]. Other key operations such as checkpoint, rollback and determinism can be implemented at the level of the C++ library linker. Therefore, we suggest speculative execution is broadly applicable across commercial titles, and can be systematized with additional work.

10. EVALUATION

We 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. Our primary tests are on Doom 3 because twitch-based gaming is very sensitive to latency. We confirm the results with limited secondary tests on Fable 3. A summary of our 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.
- Our 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.

10.1 Experimental Setup

We 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

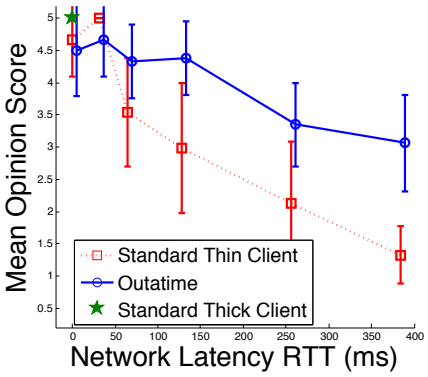


Figure 11: Impact of Latency on User Experience

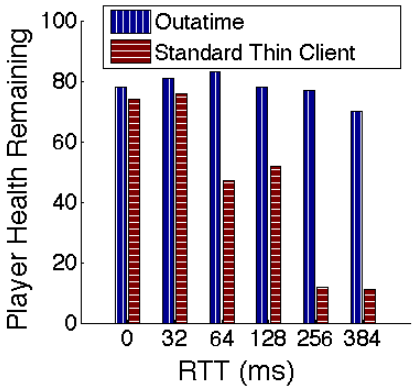


Figure 12: Remaining Health

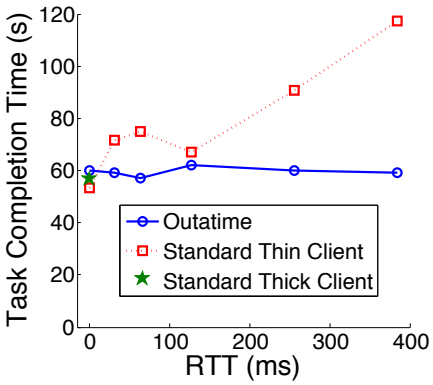


Figure 13: Task Completion Time

Figure 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, we 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. We also used the same machine to run the Thick Client. User input was issued via mouse and keyboard. We configured Doom 3 for a 1024×1024 output resolution. We 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.

10.2 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:* We 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. We 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? We also solicited free-form comments and recorded in-game vocal exclamations which turned out to be illuminating. Lastly, we recorded general player statistics during play, such as remaining player health and time to finish the level.

10.2.1 Mean Opinion Score

Figure 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.

- 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.

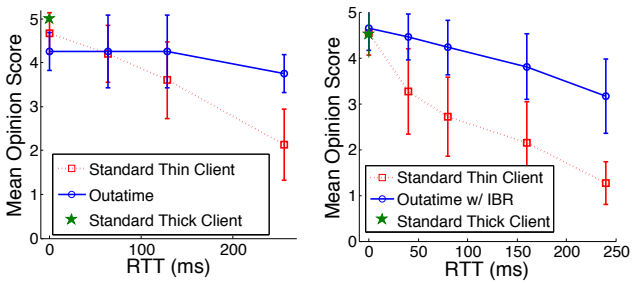
- Outatime @ 128ms: *"Controls were fluid, animations worked as expected, no rendering issues on movement and no transition issues with textures."*
- Outatime @ 128ms: *"Shooting was a bit glitchy, but you don't notice too much. Controls were about the same, good responsiveness and all."*
- Outatime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

10.2.2 Skill Impact

We found that longer latencies also hurt performance on in-game skills such as avoiding enemy attacks. For Study A, we instructed participants to eliminate all enemies in a level while preserving as much health as possible. Figure 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, suggesting that in-game skills were impaired. Outatime exhibited no significant drop off for RTT$\leq$ 256ms.



(a) Impulse Speculation (b) Navigation Speculation

Figure 14: Fable 3 MOS

	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 1: Server Processing Time Per-Frame $\pm$ StdDev. Note that Outatime's server processing time is masked by speculation whereas Thin Client's is not.

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.

10.2.3 Task Completion Time

Lastly, we measured participants' level completion time. Participants were instructed to eliminate all enemies from a level as quickly as possible. Figure 13 shows that RTT $\geq$ 256ms lowered Thin Client completion times, but had little impact on Outatime completion times.

10.3 Fable 3 Verification

We setup Fable 3 for similar testing to Doom 3. We recruited twenty-three additional subjects (age 20–34, 4 females, 19 males) who were unfamiliar with the Doom 3 experiments. In this study, we separated the testing of Impule and Navigation speculation in order to isolate their component effects. Figure 14a and Figure 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 our 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.

10.4 System Performance and Overhead

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

10.4.1 Client Performance

Figure 15 shows that Outatime and Thick Client both achieve the target frame time of 32ms, which is directly

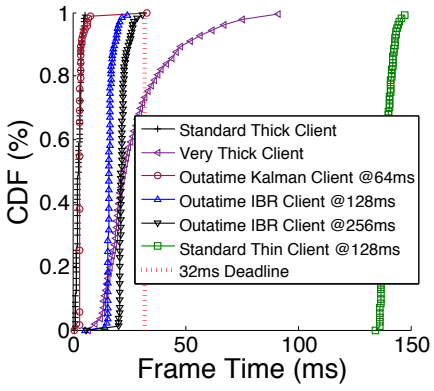


Figure 15: Client Frame Time

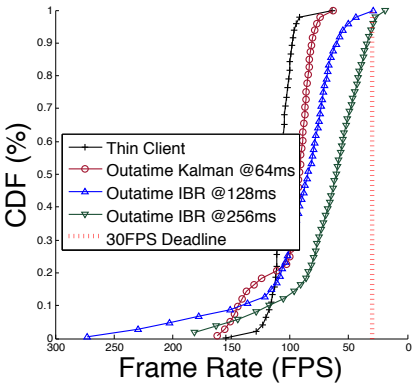


Figure 16: Frame Throughput Measured at Server

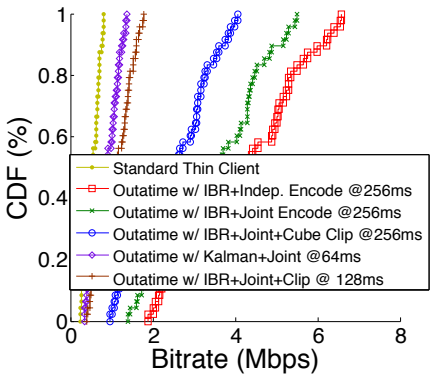


Figure 17: Bandwidth Overhead

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

Table 2: Fast Hybrid Checkpoint vs. Alternatives

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.

We 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, we 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 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 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.

10.4.2 Server Throughput and Processing Time

We next quantify the server load in two ways. The first is frame rate where we 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 our server implementation is highly parallelized.

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

Figure 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 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.

10.4.3 Checkpoint and Rollback

We perform a comparative benchmark of our fast checkpoint/rollback implementation to two other possible implementations in Table 2. The setup is a standalone stress test on Doom 3 where checkpoint speed is isolated from other system effects. Our 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.

10.4.4 Bitrate

While our prototype performs cube clipping directly on rendered frames and uses a hardware accelerated codec pipeline for efficiency, we conducted compression testing offline using `ffmpeg` and `libx264`, two publicly available codec libraries, for easier repeatability. Figure 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× the bitrate of Thin Client and a 40% reduction from independent. Finally, when RTT≤ 128ms, joint encoding and clipping consumes only 1.04Mbps, which is only 1.97× more than Thin Client. The savings are due to lower prediction error over a shorter time horizon (Figure 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× Thin Client.

10.5 Public Deployment

We deployed Outatime publicly at MobiSys 2014's demo session [27]. Around fifty participants played the system. The response was positive with best demo recognition received.

11. RELATED WORK

Speculative execution is a general technique for reducing perceived latency. In the domain of distributed systems, Crom [29] 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 [43]. The authors of [26] 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 [24] 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. [42] 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 [13, 20], the benefits of cloud-hosting and Outatime (instant start, easier management, etc.) still apply. For mitigating network latency stemming in multiplayer gaming, see §8.

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 [34]. The authors of [41] apply hardware-supported IBR to reduce local scene rendering latency. The authors of [28] 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.

12. 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 [36]. 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.

We 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 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.

13. CONCLUSION

Games, by their very nature of being virtual environments, are well-suited for speculation, roll back and replay. We 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 with demanding response times. This leads us to be optimistic about Outatime's applicability across genres, with potentially bigger savings for slower-paced games. We found that players overwhelmingly favor Outatime's masking of high RTT times over naked exposure to long latency. In turn, this enables cloud gaming providers to reach much larger mobile community while maintaining a high level of user experience.

14. REFERENCES

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

[6] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. 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.

[7] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM'08*, pages 389–400, New York, NY, USA, 2008. ACM.

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

[9] K.-T. Chen, P. Huang, and C.-L. Lei. How sensitive are online gamers to network quality? *Commun. ACM*, 49(11):34–38, Nov. 2006.

[10] M. Claypool, D. Finkel, A. Grant, and M. 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.

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

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

[13] J. R. Douceur, J. Elson, J. Howell, and J. 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.

[14] 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/.

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

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

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

[18] 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, 2014.

[19] 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.

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

[21] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. 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.

[22] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. 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.

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

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

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

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

[27] K. Lee, D. Chu, E. Cuervo, J. Kopf, A. Wolman, and J. Flinn. Demo: Delorean: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. MobiSys '14, 2014.

[28] W. R. Mark, L. McMillan, and G. 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.

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

[30] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst.*, 24(4):361–392, Nov. 2006.

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

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

[33] P. Quax, P. Monsieurs, W. Lamotte, D. D. Vleeschauwer, and N. Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In W. chang Feng, editor, *NETGAMES*, pages 152–156. ACM, 2004.

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

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

[36] M. Stanton, B. Humberston, B. Kase, J. F. O'Brien, K. Fatahalian, and A. Treuille. Self-refining games

using player analytics. *ACM Trans. Graph.*, 33(4):73:1–73:9, July 2014.

[37] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. 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.

[38] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2011.

[39] 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-gaming-service/`.

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

[41] J. Torborg and J. 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.

[42] B. Wester, P. M. Chen, and J. Flinn. Operating system support for application-specific speculation. In *EuroSys'11*, pages 229–242. ACM, April 2011.

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

[44] 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.