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

Kyungmin Lee[◇] David Chu[†] Eduardo Cuervo[†] Johannes Kopf[†]
Sergey Grizan[‡] Alec Wolman[†] Jason Flinn[◇]
[◇]Univeristy of Michigan [†]Microsoft Research [‡]Siberian Federal University

Abstract

Gaming 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 250ms of network latency. Outatime produces speculative rendered frames of future possible outcomes, delivering them to the client one entire RTT ahead of time; clients perceive no latency. To achieve this, Outatime combines: 1) future input prediction; 2) state space subsampling and time shifting; 3) misprediction compensation; and 4) bandwidth compression.

To evaluate the prediction and speculation techniques in Outatime, 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 overwhelmingly 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.

1 Introduction

Gaming is a popular activity. 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 devices such as down- and mid-market phones, basic tablets, TVs and other displays lacking high-end GPUs. Second, with cloud gaming, developers avoid two long-standing challenges that arise with the vexing diversity of devices: platform compatibility headaches and per-platform performance tuning [35, 30, 24]. Third, 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 and Amazon are among the providers that have released or announced cloud gaming services [1, 2, 3].

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 100 ms [10, 25, 6]. A further delay degradation from 150 ms to 250 ms lowers user engagement by 75% [9].

One way to address latency is to move servers closer to clients. 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 600 ms, 300 ms and 400 ms, respectively [16, 15, 27]. In fact, even well-established residential wired last mile links tend to suffer from latencies in excess of 100ms when under load [28, 5]. Unlike non-interactive video streaming, buffering is not possible for interactive gaming.

Instead, we propose to mitigate wide-area latency via speculative execution. We present *Outatime*,[1] a system that delivers real-time gaming interactivity as fast as traditional local client-side execution, despite latencies up to 250 ms. Outatime's basic approach combines input prediction with speculative execution to render mulitple possible frame outputs which could occur RTT milliseconds in the future. Outatime employs the following techniques to accomplish this.

Future Input Prediction: Given the user's historical tendencies and recent behavior, we show that some categories of user actions are highly predictable. We develop

[1]*Outatime* : a car so fast that it can time travel, enabling one to take action in the past based on possible futures.

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 constructing a Kalman filter to improve users' perception of smoothness.

State Space Subsampling and Time Shifting: 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 frames over long RTTs can be very large due to state space explosion. To address this, we use two techniques: state space subsampling, and event stream time shifting. These greatly reduce possible outcomes with minimal impact on the quality of interaction, thereby permitting speculation within a reasonable budget.

Misprediction Compensation: When mispredictions occur, Outatime enables the client to execute *error compensation* on the (mis)predicted frame. The resulting frame is very close to what the client ought to see. Our misprediction compensation uses *view interpolation*, a graphics technique that transforms pre-rendered images from one viewpoint to a different viewpoint using only a small amount of additional 3D metadata. Furthermore, to prevent past prediction errors from propagating forward, Outatime uses checkpoint/restore to recover from speculative state.

Bandwidth Compression: The transmission of possible outcome frames from server to client consumes added bandwidth. To reduce this overhead, we develop a video encoding scheme which provides better compression than standard encoding by taking advantage of the visual similarity of speculated frames.

To punctuate our emphasis on fast interaction, we evaluate Outatime's prediction techniques using two *fast 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 mobile games in the first person shooter and role playing genres, respectively.

Through interactive gamer testing, we found that players perceived only minor differences in responsiveness on Outatime when operating at up to 250 ms RTT when compared head-to-head to a system with no latency. Moreover, unlike in standard cloud gaming systems, Outatime players' in-game skills performance and task completion times did not drop off as RTT increased up to 250 ms. Overall, player surveys indicated positive reception of gameplay on Outatime. Speculation's latency reduction benefits do come with a cost. We show that while several of our compression techniques are able to dampen increased bandwidth costs, Outatime exhibits a bitrate that is a factor of $1.5-4.5\times$ higher than standard cloud gaming systems. On the whole, we believe this is a reasonable trade-off for service providers who are otherwise unable to offer users low-latency interactivity.

The remainder of the paper is organized as follows. §2 provides background on 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. §7 discusses how we reduce bandwidth overhead. §8 covers the implementation. §9 evaluates Outatime via user study and performance benchmarks. §10 covers related work and §11 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.

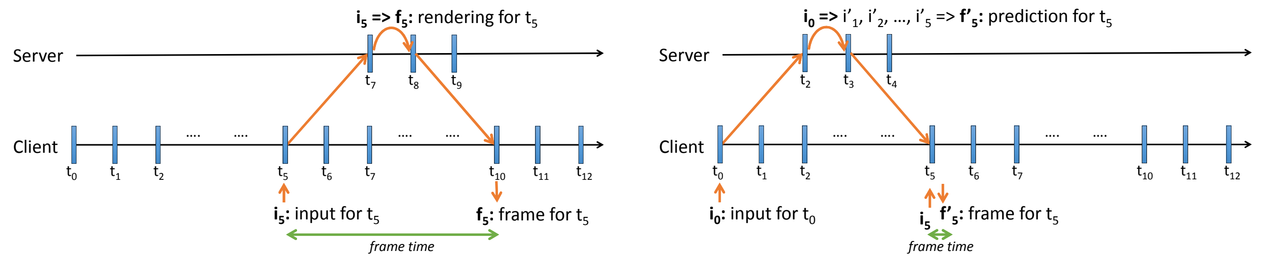
Network latency has an acute effect on interaction for cloud gaming. In standard cloud gaming, the frame time must include the additional overhead of the network RTT, as illustrated in Figure 1a. Let time be discretized into 32ms clock ticks, and let the RTT be 4 ticks (128ms). At t_5, the client reads 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 and large jitter. In exchange, we are willing to transmit a higher volume of data and potentially even introduce (very small and very 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 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,

[2] $\frac{1}{30fps} \approx$ 32ms 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
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

(a) Standard cloud gaming: Frame time depends on net latency. **(b)** Outatime: Frame time is negligible.

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

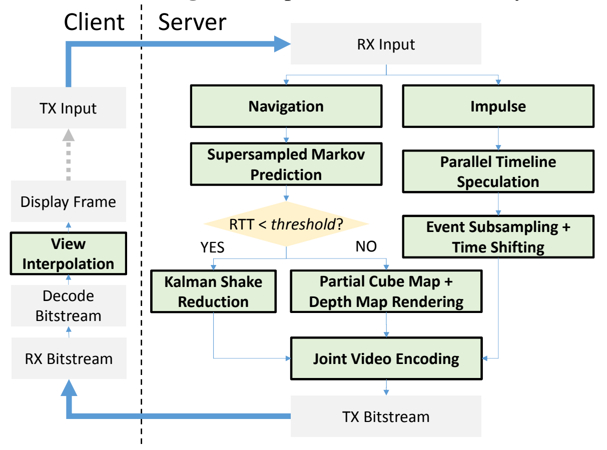


Fig. 2: The Outatime Architecture. Bold boxes represent the main areas of this paper's technical focus.

computes a sequence of probable future input up to one RTT later as $i'_1, i'_2, \ldots, i'_5$ (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: $i_1 = i'_1, i_2 = i'_2, \ldots, i_5 = i'_5$. 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 [33]. If the input sequence differs, the client applies *error compensation* to f'_5 and displays a corrected frame. We describe error compensation in detail in §4. 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 example of delay tolerant input in Doom is the command that shows 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 that Outatime, like standard cloud gaming systems, makes minimal assumptions about client capabilities. Namely, the client must perform standard operations such as decode a bitstream, display frames and transmit standard input such as button, mouse, keyboard and touch events. An additional requirement for Outatime is that the client should be able to execute view interpolation, a misprediction compensation procedure which consists of basic graphics operations that can be performed efficiently on any device with a GPU. In contrast, high-end games that run solely on a client device demand much more powerful CPU and GPU processing.

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 experimented with more sophisticated time series models [26], including neural network time-series prediction as well as linear and polynomial regression models, yet we observed that the Markov chain performed comparably to these others for our task. 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 using an offline per-user training step. 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.

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, we discuss the issue of training in more detail.

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. We discovered that supersampling helps with prediction accuracy empirically. Our hypothesis is that supersampling provides a benefit because prediction accuracy degrades non-linearly over time. 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 §9 is shown in Figure 3. Most dimensions of rotational and translational displacement exhibit little performance degradation with longer RTTs. Yaw (θ_x) 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), per frame errors are sufficiently minor and infrequent. Note that the client can always 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.

Bootstrap Time. Construction of a reasonable Markov Model requires sufficient training data collected during an observation period. It is important that the observation period is of sufficient duration to accurately reflect the distribution of transition probabilities during extended gameplay. Otherwise, mispredictions due to inaccurate transition probabilities are severe. 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. Compared to the average player session length (which varies by game genre; for an RPG similar to Fable 3 the authors of [9] report four hours), 300 seconds is a modest bootstrap period. Currently, training is performed once per user and is independent of the game level or map.

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

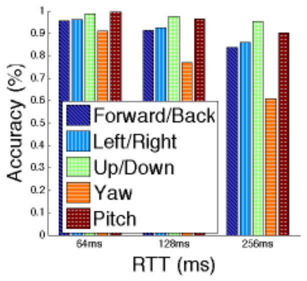


Fig. 3: Doom 3 Navigation Prediction Summary. Roll (θ_z) is not an input in Doom 3 and need not be predicted.

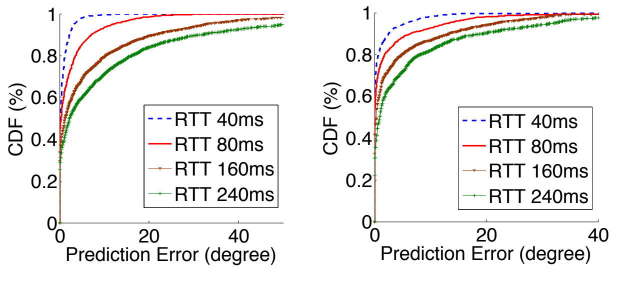


Fig. 4: Prediction for Yaw (θ_x), the navigation component with the highest variance. Error under 4° is imperceptible.

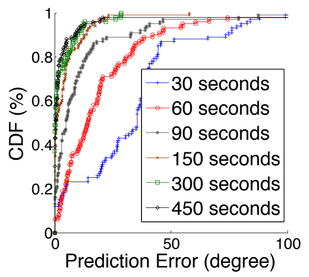


Fig. 5: Error Decreases with More Observation Time. Data is for Fable 3 at RTT= 160ms.

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 [19] 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. Before and after video clips at `http://1drv.ms/1koGZ1p` show that shake is largely eliminated by Kalman filter.

Misprediction Compensation with View Interpolation. 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'.

View Interpolation. We compensate for mispredictions with *view interpolation*. View interpolation was originally developed as a means to derive novel camera viewpoints from a fixed number of initial cameras. It operates by having initial cameras capture depth information (f^{Δ}) in addition to 2D RGB color information (f'). It then interpolates to create a new 2D image (f'') from f' and f^{Δ} [29]. 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

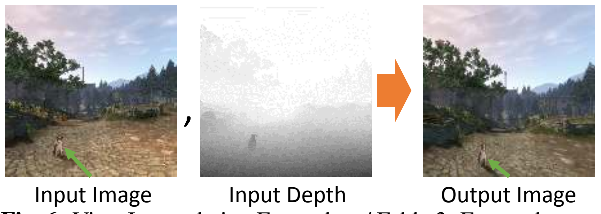


Fig. 6: View Interpolation Example w/ Fable 3. Forward translation and leftward rotation is applied. The dog (indicated by green arrow) is closer and toward the center after interpolation.

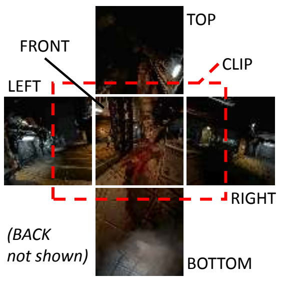


Fig. 7: Cube Map Example w/ Doom 3. Clip region shown.

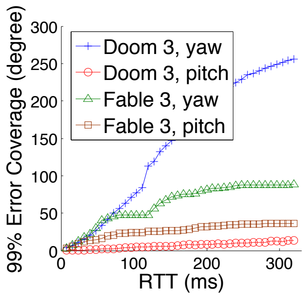


Fig. 8: Angular coverage of 99% of prediction errors is much less than 360° even for high RTT.

of blurred pixels when interpolation is inaccurate.

For effective interpolation, 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 interpolated view is bounded within the original. To handle this case, instead of rendering a normal 2D image by default, we render a cube map [13] 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 interpolated image is within its bounds. A video clip demonstrating view interpolation is at `http://1drv.ms/1kpb5lp`.

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. We use the following technique to reduce this overhead.

Clipped Cube Map. For the 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 $1-\varepsilon$. 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 $1-\varepsilon$ of the expected probability density:

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

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 $1-\varepsilon$ 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 $\varepsilon = 0.01$, meaning that 99% of mispredictions are compensated. Doom 3's pitch range 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.

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

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.

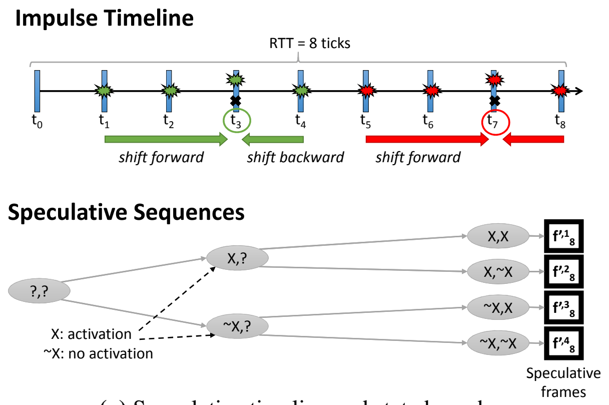
The objective for impulse speculation is to respond quickly to player's impulse input while avoiding any visual inconsistencies. For example, in a first person shooter, weapons should fire quickly when triggered, and enemies should not reappear shortly after dying. The latter type of visual (and semantic) inconsistency is disconcerting to players, yet may occur when mispredictions occur in a prediction-based approach. Therefore, 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.

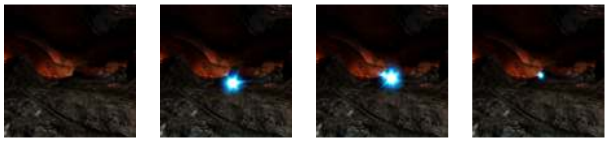
As RTT increases, the number of possible sequences grows exponentially. Consider an RTT of 256ms, which is 8 clock ticks. An activation may lead to an event registration at any of the 8 ticks, leading to an overwhelming 2^8 possible sequences. In general, 2^λ sequences are possible for an RTT of λ ticks.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

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 be counter-intuitive to users.

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 se-



(a) Speculative timeline and state branches



(b) ~X, ~X **(c)** ~X, X **(d)** X,~X **(e)** X,X

Fig. 9: 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.

quence 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 9a 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 9a shows the possible event sequences and four resulting speculative frames, $f_8'^1$, $f_8'^2$, $f_8'^3$ and $f_8'^4$. 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 [25, 6], 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 9a. 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.

Ternary and Quaternary 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.

Delay Tolerant Events. We classify any input event that is slow relative to likely RTTs 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 any subsequent 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 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 *state coordination game server* which reconciles distributed state updates to produce an eventually consistent view of events. For responsiveness, each client may perform local dead reckoning [12, 8]. 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 to remedy glitches. Techniques that selectively process other players as AI-controlled (and thereby deterministic) opponents may be insightful in mitigating state space explosion [7].

7 Bandwidth and Encoding

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* similarity. Using Figure 9a 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 9b–9e show that the surrounding environment is largely unchanged, and therefore the spatial similarity 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 similarity is also often high. Similarly, navigation speculation's clipped cube map faces often exhibit both temporal and spatial similarity.

Outatime takes advantage of temporal and spatial similarity 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, 17]. These hardware accelerated capabilities, which otherwise sit idle, are reprogrammable for our speculation's joint encoding.

8 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 and suggests that the essential developer modifications needed to support efficient speculation are similar across commercial titles. We also examined UDK [12], one of several widely used commercial game engines upon which many games are built, and verified that the modifications described below are general and feasible in UDK as well.[4] Therefore, we suggest that the techniques proposed below are broadly applicable and can be systematized.

Doom 3 is structured as a frontend executable, known as the game engine, and a content library. The Doom 3 engine (also known as idTech4), `doom3.exe`, performs generic game routines such as capturing user input and rendering graphical content. The Doom 3 content library, `gamex86.dll`, involves everything specific for the game running on top of the engine, in this case Doom 3. The content library is responsible for performing game specific logic and handling the simulation of the game state. As a preliminary step to permit deterministic replay, we

[4] UDK source was only publicly released recently in April 2014.

made changes according to [33] such as de-randomizing the random number generator.

We have made the following key modifications to Doom 3. To support impulse speculation, we spawn up to four Doom 3 *slaves*, each of which is a modified instance of the original game (i.e., `doom3.exe` and `gamex86.dll`). 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: `rendervi`, which produces the cubemap and depth maps necessary for interpolation. The number of slaves spawned depends on the network latency. When RTT$>$ 128ms, four slaves can cover four speculative state branches. Otherwise, three slaves suffice. The client is a simple thin client with the ability to perform view interpolation [29].

As with other systems that perform speculation [22, 32], 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. As a point of comparison, the built-in save/load game “checkpoint” feature takes 20 seconds, which would yield 0.05fps.

Unique among speculation systems, we use a combination of page-level checkpointing and object-level checkpointing. This is because whereas page-level checkpointing is application agnostic and 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 *Game State Objects (GSOs)*: those non-constant objects which reproduce the world state. Checkpointing objects which have no bearing on the game state or are constant, such as already converted raw user input data or stateless rendering handlers, only cause runtime overhead. `gamex86.dll` consists almost exclusively of GSOs whereas `doom3.exe` has a mix of GSOs and other objects for handling user input and output. Therefore, we use object-level checkpointing for `doom3.exe` and page-level checkpointing for `gamex86.dll`.

To implement page-level checkpointing for `gamex86.dll`, 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 9a, 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 allocaltor 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.

To implement object-level checkpointing for `doom3.exe`, we track lifetimes of object rather than pages. Conveniently, `doom3.exe` objects are stateless, and therefore checkpoint bypasses saving state with copy-on-write. To discard a speculation, we delete any object that did not exist at the checkpoint, and restore any objects that were deleted during speculation.

We implemented the server-side joint video encode pipeline and client-side decode pipeline as Nvidia CUDA kernel functions executing on the GPU with support from dedicated codec accelerators [23]. 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’s view interpolation as an OpenGL GLSL shader which consumes decoded raw frames and produces a compensated final frame for display. For any reasonable interactive performance target, CPU processing of any of the above steps is infeasible since the PCI-E bus is easily saturated during high frequency data transfer of uncompressed video frames between GPU and CPU. Moreover, codec processing and view interpolation are inherently parallel and therefore well-suited for the GPU.

9 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 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, users rate Outatime’s

impulse speculation playable with minor responsiveness impairment up to 256ms.
- Users rate Outatime's navigation speculation playable with minor visual quality impairment up to 256ms.
- Users experience very little in-game performance degradation with Outatime as compared to a standard cloud gaming system.
- Speculation imposes increased demands on resource. Bandwidth consumption is $1.5 - 4.5\times$ higher than standard cloud gaming depending on the RTT.

Experimental Setup. We tested Outatime against the following baselines. *Standard Fat 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 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 server 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 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 Fat Client. User input was issued via mouse and keyboard. We configured Doom 3 for a 1024×768 output resolution.

We divided the user study evaluation into an assessment of impulse and navigation speculation in order to precisely assess the impact of each. Twenty three participants consisting of coworkers and colleagues were recruited based on their interest in a call for participation in a gaming study. No compensation was offered. All participants except one were males. The age range was 24 – 42. Prior to engagement, they 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 participants reported either Beginner (score=2) or No Experience (score=1) for Doom 3, participants exhibited a range of overall and genre-specific skill levels from Expert (score=5) to Beginner, with an average self-assessment of Experienced (score=4).

Impulse Speculation Performance. We evaluated Impulse Speculation according to 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 fat 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 eight 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 fat client as a control. Participants 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 player health, enemies eliminated and time to finish the level.

Mean Opinion Score. Figure 10 summarizes overall MOS when playing on Outatime, Thin Client and Fat Client at various RTTs. Fat client is not MOS= 5 due to a placebo effect. Thin Client MOS follows a sharp downward trajectory, indicating that the game becomes increasingly frustrating to play as early as 128ms. Free form participant comments 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 @ 256ms: *"I hated it. Too difficult to play. It overreacts."*

For Outatime, the MOS stays relatively high with scores between 4 to 4.5 up through 256ms, with a slight drop off at 384ms. Comments are shown below.

- Outatime @ 256ms: *"I think I was playing the original. If it was not the reference, that was a good one."*
- Outatime @ 384ms: *"A little delay. Not annoying."*

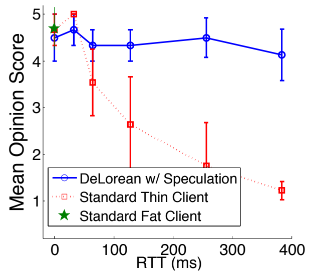


Fig. 10: Impulse Speculation

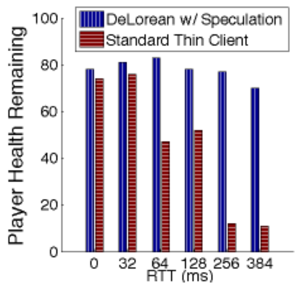


Fig. 11: Remaining Health

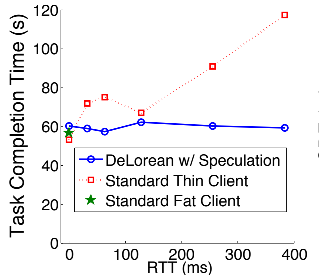


Fig. 12: Task Completion Time

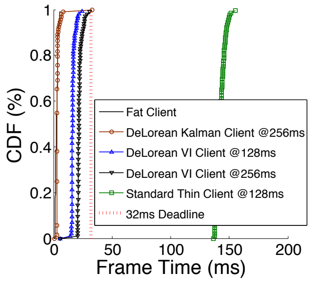


Fig. 13: Client Frame Time

Overall, players with high prior experience ("expert" or "experienced") tended to assign lower MOS ranks though it was not statistically significant. Responsiveness MOS ratings were very similar to overall MOS ratings. Ratings for visual quality were similar, which was expected since impulse speculation does not introduce visual artifacts. The results are elided for space.

Skill Impact. We found that longer latencies also hurt performance on in-game skills such as avoiding enemy attacks. We instructed participants to eliminate all enemies in a level while preserving as much health as possible. Figure 11 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.

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.

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 12 shows that RTT $\geq$ 256ms lowered Thin Client completion times, but had little impact on Outatime completion times.

Navigation Speculation Performance. To evaluate the speculation performance for navigation, we again used MOS. Because the visual differences were often subtle, we had participants watch recorded traces of either their own or other participants' gameplay. Each participant was presented videos for each latency setting {0ms, 64ms, 128ms, 256ms, 384ms}. At each latency, we tested: 1) navigation speculation with view interpolation, 2) with Kalman shake correction, and 3) with neither view interpolation nor Kalman correction. Videos were shown side-by-side with the Reference so that it was easier for participants to spot differences. Lastly, we also included a control of Standard Fat Client. Overall, participants assessed videos of two different settings: *exploration mode* without enemies present, and *combat mode* with enemies. Figure 14 shows participants' MOS scores. For Outatime w/ view interpolation in exploration mode, user experience is above 4.5 up until RTT= 256ms. In combat mode, MOS decreases sharply past 256ms. This is due to the visual artifacts appearing near enemies. In comparison, Outatime w/ Kalman performs somewhat reasonably at low RTT= 64*ms* across both settings, and therefore is useful when saving bandwidth is important (as we show in the following section), but is noticeably worse for many high RTT settings, ranking just above non-annoying most of the time. Lastly, Outatime w/o Kalman does not perform well due to excessive video shake.

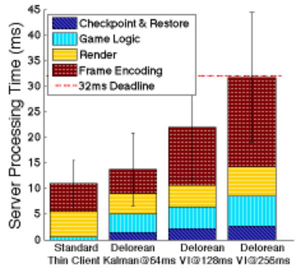


Fig. 16: Server Processing Time Breakdown Per Frame

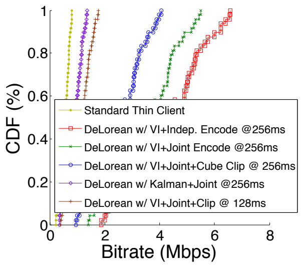
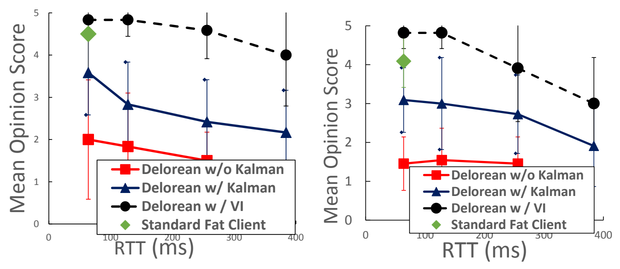


Fig. 17: Bandwidth Overhead

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. Figure 15a and Figure 15b show that the MOS impact of impulse speculation and navigation speculation are similar in Fable 3.

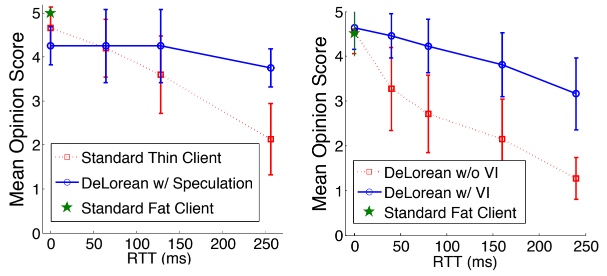
System Performance and Overhead. We measure client, server and bandwidth utilization. During server testing, we use a trace-driven client for repeatability. Similarly, we use a trace-driven server during client tests.

Client Performance. In Figure 13, Outatime and Fat Client both achieve the target frame time of 32ms, which directly leads to players' perceptions of low latency. The bulk of the time is spent on decoding, which we have not optimized. View interpolation accounts for $<$ 2ms, even when run on a 2010 notebook's Nvidia GT 320M, which is 21$\times$ less powerful than the GTX 680. In contrast, thin



(a) Exploration **(b)** Combat

Fig. 14: Navigation Speculation



(a) Impulse Speculation **(b)** Navigation Speculation

Fig. 15: Fable 3 MOS

Client's frame time is clearly vulnerable to RTT.

Server Utilization. We quantify the server load in Figure 16. Outatime with Kalman incurs overhead on top of Thin Client due to impulse speculation, checkpoint, restore and rollback. Outatime with View Interpolation brings further overhead due to cube and depth map rendering and frame transfer. Overhead is higher at 256ms than at 128ms due to the need to run extra slaves to service more speculative branches. At 256ms, server processing time is sufficient on average to keep up with a 32ms frame time.

Bitrate. While our prototype uses hardware accelerated codec pipeline for efficiency, we conducted compression testing 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 View Interpolation-based 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. 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 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.

10 Related Work

Speculative execution is a general technique for reducing perceived latency. In the domain of distributed systems, Crom [21] 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 [34]. The authors of [20] 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 [18] 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. [32] 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.

We share a similar goal with the authors of [31] in aiming to reduce latency for cloud-hosted gaming. Their complementary approach looks at adapting bitrate to available bandwidth. Additional efforts to mitigate network latency for multiplayer games are discussed in §6. Alternative app distribution avenues such as HTML5 are intriguing, though interactive games have had stronger performance demands than what current browsers offer. Even with native client execution [11, 14], the benefits of cloud-hosting and Outatime still apply.

11 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 the work of applying Outatime to other genres. 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 a much larger community while maintaining a high level of user experience.

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. 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, and A. Juditsky. Nonlinear black-box modeling in system identification: a unified overview. *Automatica*, 31(12):1691–1724, 1995.

[27] 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.

[28] 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.

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

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

[31] S. Wang and S. Dey. Addressing response time and video quality in remote server based internet mobile gaming. In *WCNC*, pages 1–6, 2010.

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

[33] D. Who. Citation anonymized.

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

[35] 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.