Prime: A Framework for Co-located Multi-Device Apps

David Chu[†] Zengbin Zhang[‡] Alec Wolman[†] Nicholas Lane[◇]
[†]Microsoft Research [‡]UC Santa Barbara [◇]Bell Labs
davidchu@microsoft.com zengbin@cs.ucsb.edu alecw@microsoft.com niclane@acm.org

ABSTRACT

Even though mobile devices are ubiquitous, the conceptually simple endeavor of using co-located devices for multi-user experiences is cumbersome. It may not even be possible when certain apps are not widely available.

We introduce Prime, a thin-client framework for co-located multi-device apps (MDAs). It leverages well-established remote display protocols to enable spontaneous use of MDAs. One device acts as a host, executing the app on behalf of connected clients.

The key challenges is dynamic scalability: providing high framerates, low latency and fairness across clients. Therefore, we have developed: an online scheduling algorithm that provides frame rate, latency and fairness guarantees; a modified 802.11 MAC protocol that provides low-latency and fairness; and an efficient video encoder pipeline that offers up to fourteen times higher framerates. We show that Prime can scale a host up to seven concurrent players for a commercially released open source action game, achieving touch-to-pixel latency below 100ms for all clients.

ACM Classification Keywords

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

Author Keywords

Thin client computing; Mobile resource scheduling

1. INTRODUCTION

Mobile devices are ubiquitous, and we often encounter not only our own devices, but also those of family, friends and new acquaintances. What if any app had the capability to instantaneously span all surrounding devices? For example, a student might invite her classmates to share in an interactive pedagogical music demonstration [46]. Colleagues might collaboratively share and edit photos [25, 7]. A group of friends might explore a new area through the lens of a game on their mobile device [2, 4, 5]. Fellow commuters might pass time gaming together [45, 6]. Many such *co-located Multi-Device Apps (MDAs)* have been prototyped in the research community.

Recent advances such as Wi-Fi Direct, Bluetooth Low Energy and NFC make it very easy for mobile devices to discover each other and communicate directly over short range wireless links. With support for Miracast, AirPlay, Chromecast and other similar protocols in recent operating systems, manufacturers recognize users' desire to harness co-located devices. These protocols enable basic one-to-one streaming of passive content.

Beyond basic one-to-one screen streaming, a much more interesting and general case is within reach: MDAs can (1) enlarge the number of participating devices from two to n, and (2) promote active interaction through any device, instead of merely passive consumption.

One way to initiate such an MDA is to require every co-located device to install the MDA from an app store. However, devices run a mix of platforms (Android and its various forks, iOS, Windows, etc.) and much anecdotal evidence suggests that developers find it challenging to support multiple platforms. We examined this issue empirically and found that the likelihood of a popular app from store X existing in store Y tends to be less than 50% and can be as low as 5% (§2). This situation is especially dire for new mobile platforms trying to break in to the market dominated by Android and iOS. Our findings suggest that MDAs which rely on app stores are likely to lock out potential MDA participants.

Our system Prime[1] enables MDAs to achieve fluid interaction among n devices. Prime bypasses platform heterogeneity issues by using a remote display protocol, a virtualization technique where clients relay input events to the server, and then display an audio/video stream from the server. In Prime, a *host device* runs application instances on behalf of nearby *client devices*. To join an MDA, clients merely need to have installed a remote display client app once. Unlike other means of MDA establishment (*e.g.,* [29, 28]), clients need no per-app software. With such minimal assumptions, Prime enables spontaneous multi-device interactions.

While the basic building block of remote display is well-studied [1, 20, 21, 34, 44, 23], scaling a mobile MDA to even a moderate number of devices poses unique challenges because requirements on *frame rate*, *response latency* and *fairness* must be satisfied while running on a resource-constrained host device and wireless channel.

UbiComp '15, September 7–11, 2015, Osaka, Japan.

http://dx.doi.org/10.1145/2750858.2806062

[1] Proximity Remoting for Instantaneous Multi-device Execution

Frame rate and response latency are important for interactive MDAs. Frame rate below 30 frames per second (fps) or response latencies over 100ms cause user dissatisfaction for interactive multimedia apps and games [3, 10] For latency, we focus on end-to-end *touch-to-pixel* latency, which is the latency between client input and a correspondingly updated client screen. Fairness is particularly pertinent for competitive MDAs like games.

To address these challenges, we have built Prime to optimize MDAs for frame rate, latency and fairness. It consists of the following components. First, we design a lightweight *Host Scheduler* that is able to maintain stochastic guarantees on end-to-end latency, throughput and fairness even when scheduling across heterogeneous subcomponents such as CPU, GPU, codec accelerators and the wireless channel. Moreover, we show empirically that it can more efficiently utilize such resources by up to $1.4-2.3\times$ as compared to standard implementations.

Second, we design a *Wireless Scheduler* that modifies the wireless MAC to lower latency and improve fairness. This is needed because we show 802.11 is vulnerable to certain *priority inversions*, leading to user input delays and reorderings, and inhibiting both responsiveness and fairness. In response, we demonstrate an 802.11-compatible MAC that overcomes this problem to provide a 95th percentile input transmission latency of 3ms.

Third, we construct a video *Encoder Pipeline* that uses hardware acceleration and eliminates data copy overhead to improve frame rate by a factor of up to $14\times$ as compared to a software-only implementation.

While these improvements can help any potential MDA, we demonstrate these benefits on *MarbleMaze*, a commercially released open source game. We picked a game because they are highly popular [13], and naturally place emphasis on good frame rate, low latency and fairness. With Prime, MarbleMaze can support up to seven concurrent users with a median latency of 54ms, and latency divergence among users of no more than 6.5 ms. This compares favorably to a basic remote display implementation which supports only three users with unacceptably high variable response times.

2. MDAs USING EXISTING APPROACHES

Two seemingly reasonable avenues for enabling MDAs are existing app stores and existing remote display implementations. We first examine why these fall short.

2.1 App Store Analysis

A cursory inspection might suggest existing app stores are sufficient for enabling MDAs. However, while commonplace, today's app stores are fragmented, and there are both technical and non-technical barriers that stop app developers from distributing their apps on multiple platforms. In this section, we confirm this observation empirically.

We first quantify the likelihood of apps *cross-listing*, *i.e.,* appearing in multiple stores. This signals whether a given app is even available across platforms. We considered four major US app stores: Apple iTunes Store, Google Play, Amazon App Store and Windows Phone Store. We scoped our investigation to top-ranked apps in the gaming category because these are popular and are often multiplayer.

From store...	Top-k	... existing in another store		
		Google	Apple	Windows
Google	500	–	70%	5%
Apple	4320	42%	–	10%
Windows	12300	15%	20%	–
Amazon	21388	49%	–	–

Table 1. App Cross-Listing Frequency. Top-k is number of apps crawled from a given store.

In particular, we crawled 4320 games from Apple iTunes Store, 500 from Google's Play store, 21388 from Amazon's App Store and 12300 from Microsoft's Windows Phone Store between September and October 2014. From this data, we randomly selected a subset of apps and performed existence checks: queries against other app stores for an app's existence to compute the likelihood that the app is cross-listed in other stores. Matches were based on loose app title string comparison and manual verification. Due to API restrictions, we were not able to perform cross-listing checks against the Amazon store, nor were we able to crawl more than 500 Google apps nor more than 4320 Apple apps. Note that an existence check is a query against a store's entire database, not merely the top-k that we crawled.

Table 1 shows the cross-listing percentages. Despite the fact that our analysis is based on the top-k most popular apps (which receive more developer attention than less popular apps and are therefore more likely to be cross-listed), the probability that an app shows up in another store is typically below 50%. The highest ratio is 70% when searching Google Play games in iTunes store, which is due in part to the fact that our Google Play dataset is limited to the top 500 apps. Note that even in this small top-500 set, there are still 150 games that do not exist in the iTunes store. Another thing we found interesting is the unexpectedly low ratio of Amazon Apps in the Google Play Store. Google's Android and Amazon's fork are nearly binary compatible, so there is almost no effort required to deploy an app to both stores. Yet developers are only doing so at a rate of 49% which is surprisingly low. Finally, a user that uses a less popular platform like Windows Phone is unlikely to be able to use apps found on popular platforms (5-20%). This makes it especially difficult for new platforms with new innovations like Firefox OS to prosper.

Based on the above data, we also assessed how probable it is for n co-located users to be able to download the same game from their corresponding app stores. Given the current market share of mobile OSes,[2] with five co-

[2] `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`:
Android:84%, iOS:12%, Windows:3%, Others:1%

located users, it is more likely that the MDA instantiation will fail (53%) rather than succeed; larger groups are even less likely. Note that this is a conservative estimate, since our analysis is based on only top-k apps. These results suggest that the discrepancies among app store catalogs are significantly large, and can seriously limit deployment of MDAs.

2.2 Lessons from a Remote Display Strawman

On the surface, existing traditional remote display implementations appear well-suited for MDAs. In fact, our initial prototype was just that; we forked a traditional implementation of remote rendering [26]. However, our early prototype revealed three important lessons.

First, traditional systems offer only best-effort latency and no frame rate guarantees because classic office worker workloads are much less sensitive to latency and frame rate variations [34]. In contrast, Prime needs to process frames fast enough (>30fps), while meeting an end-to-end touch-to-pixel latency target, λ_{e2e}, which can be broken down as follows: λ_{up}, the wireless uplink from client to host; λ_{host}, the total processing time on the host device, and; λ_{down}, the wireless downlink from host to client. While best-effort is sufficient for λ_{down} because the downlink flow is large, best-effort scheduling simply does not provide stable latency for λ_{up} and λ_{host}.

Second, traditional systems do not provide fairness amongst users (nor do they need to). In contrast, clients in close proximity to one another can quickly detect imbalanced responsiveness, particularly in competitive games (e.g., first to tap wins). Jitter in λ_{up} or λ_{host} manifests as event reordering, directly impacting the interaction outcome experienced by users. Unfortunately, standard 802.11 can easily lead to *priority inversion* where earlier messages are subject to exponential delay from later messages, causing them to be even more tardy, as we show in §5. Similarly, best-effort processor scheduling on the host can exacerbate unfairness.

Lastly, in traditional remote display settings, servers and networks are provisioned *a priori* to handle expected workloads [34]. With the ability to carefully collect workload profiles offline, budget for necessary resources, and incrementally add capacity, server administrators operate on long time scales. In contrast, we envision MDA users engaging in ad hoc groups of co-located devices where the only devices of relevance are those immediately available. Hence, scaling must happen dynamically in response to the number of users that happen to desire to join in. In such cases, best-effort scheduling suffers for additional reasons. First, best-effort is not aware of implicit dependencies among tasks. For example, network encoding depends upon rendering; delaying an earlier stages can mean later stages cannot make the deadline. Second, best-effort is not aware of implicit contention among system resources, namely the CPU, GPU and codec accelerator. Without modeling implicit contention, our early prototype suffered erratic performance drops. Third, with traditional remote rendering systems, encoding frames is very CPU- and memory-intensive, introducing large delays [34].

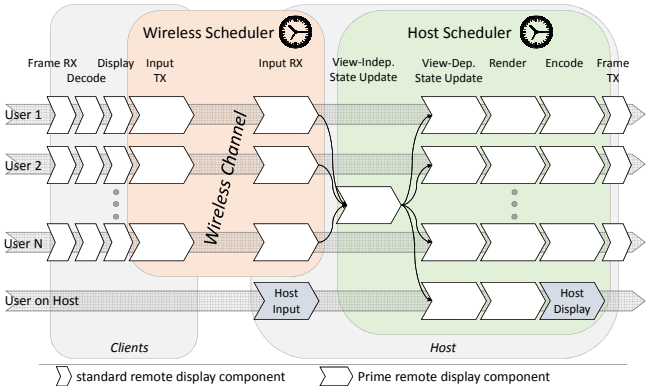


Fig. 1. The Prime Framework. Clients connect directly to the Host using Wi-Fi. Apps are executed as dataflow. Prime's Host and Wireless Schedulers optimize the dataflow for latency, throughput and fairness.

All told, our early prototype barely managed scale to three total users, routinely hitting troughs of 10fps and as much as tens of *seconds* of touch-to-pixel latency. In the following sections, we discuss our solution to these problems in detail.

3. Prime FRAMEWORK

Just like in traditional remote display, a host device runs app instances on behalf of nearby client devices. In order to overcome the limitations of traditional remote display systems, Prime introduces a dataflow programming model to relieve developers from handling scaling complexities. The Host Scheduler and Wireless Scheduler are responsible for executing the dataflow to achieve the best latency, throughput and fairness. The interplay of the dataflow and schedulers is shown in Figure 1.

For each client, Prime manages the following dataflow: (1) client input transmission and reception (λ_{up}), (2) app-specific state update, (3) frame rendering, (4) frame encoding, and (5) frame transmission back to the client (λ_{down}). Steps (2)–(4) compromise λ_{host}. App-specific state update can be divided into two stages: (a) View-Independent State Update, which takes the input of all users and updates the application logic, and (b) View-Dependent State Update, which updates the scene for each client. Developers only need to implement these two stages. To meet an end-to-end touch-to-pixel latency target of λ_{e2e}, Prime first computes λ_{down} and λ_{up} from the stochastic guarantees of the Wireless Scheduler presented in §5. The remainder $\lambda_{host} = \lambda_{e2e} - \lambda_{down} - \lambda_{up}$ is the latency optimization target provided to the Host Scheduler presented in §4.

Beyond the communication channel which is handled by the Wireless Scheduler, a participating client merely needs a traditional thin client implementation that virtualizes input, decodes frames and presents them to the client's display. Since many such client implementations exist for a wide variety of operating systems, we note that it is conceptually straightforward to reuse these for

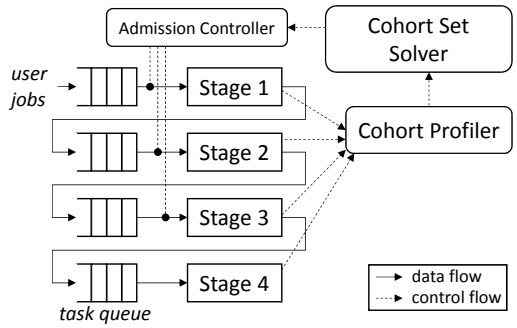


Fig. 2. Host Scheduler. Admission controller gates the number of tasks that are permitted to run. The gate policy is set by the Cohort Set Solver, which is in turn fed data by the Profiler.

Prime to demonstrate broad cross-platform support. We leave this porting effort for future work. Our focus for this paper are the Host and Wireless Schedulers.

4. HOST SCHEDULING

To consistently meet performance targets on the host, we design an online profile-driven scheduling framework, which handles dataflow applications executing on heterogeneous processors (CPU, GPU, and codec accelerators). The schedule combines a profiling step with a dynamic programming-based solution that can incrementally search for (non-obvious) performant schedules at runtime. Non-obvious yet optimal schedules often exist in practice due to surprising concurrency and hidden contention.

Concurrency- and Contention-awareness. Elementary pipelining principles would suggest scheduling one CPU task concurrently with one GPU task. However, realistic contention and concurrency conditions can be far more complicated. For example, sometimes scheduling many similar tasks at the same time can improve performance. This is due to inherent core parallelism and load-based DVFS supported by mobile CPUs, GPUs and accelerators. Contention sometimes is also subtle and implicit. For example, the CPU and GPU share the same memory bus on most SoCs; and the H.264 video codec accelerator and GPU share some of the same compute units in a partially accelerated architecture (see §6). By profiling for concurrency benefits and contention problems, Prime's host schedule is able to guarantee a stable state frame rate with high throughput, low latency and fairness amongst users.

Overview and Nomenclature. We define a *job* as the host's work to generate one frame of output for one user. Users submit job requests to the scheduler in round robin order. Requests may include input if the host received new client input since the last request was started. A job consists of multiple *tasks* where each task is the work performed by one dataflow stage in Figure 1, excluding input RX and output TX. Within a job, the task at stage $i+1$ is dependent upon completion of the task at stage i. Note that a single user may have multiple jobs in flight at the same time e.g. at different stages of the dataflow.

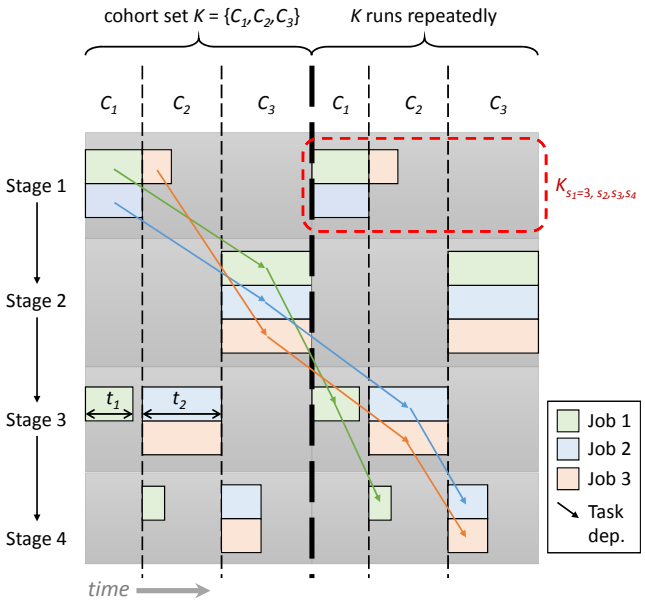


Fig. 3. Cohort Scheduling Example, $n = 4$ stages. The cohort set K which starts $j = 3$ jobs consists of cohorts $C_1 = \langle 2,0,1,0\rangle$, $C_2 = \langle 1,0,2,1\rangle$ and $C_3 = \langle 0,3,0,2\rangle$. Two sequential executions of K are shown.

The Host Scheduler employs a *cohort*-based approach. A cohort is a set of tasks which are scheduled all at once, and no other tasks are scheduled until all the tasks within the cohort are finished. In other words, a cohort is scheduled with mutual exclusion relative to all other cohorts. The rationale behind cohorts is that they can be independently profiled and combined into larger cohort sets with readily analyzable throughput properties. Tasks in successive cohorts satisfy the dependencies and thus form a valid processing stream. Our algorithm searches for the best cohort set such that the tasks can make best use of the resources, achieving concurrency and avoiding contention. The cohort setting is adapted over time to handle task variations.

Specifically, our scheduler has two components, a Profiler and a Solver as shown in Figure 2. The Profiler empirically profiles the dataflow during runtime to monitor task completion time as a function of concurrency (§4.1). The Solver uses profiling data and dynamic programming to construct a schedule that optimizes throughput as well as the response time (§4.2). The Solver also assigns users to jobs so as to ensure fairness across users.

4.1 Profiling

The Profiler evaluates the performance for each potential composition of a cohort of tasks, and feeds this information into the Solver for optimization. Let a cohort $C = \{s_i\}_{i=1..n}$ consist of s_i tasks at stage i with a total of n stages. Using Figure 3 as a running example, cohort C_1 consists of three tasks: $\langle s_1 = 2, s_2 = 0, s_3 = 1, s_4 = 0\rangle$, meaning this cohort has two concurrent stage-1 tasks and one stage-3 task. Profiling returns the finishing time of the cohort $f(C)$ as the time it takes for all tasks in the cohort to complete. For example, in Figure 3 t_1 is less

than C_1's finishing time whereas t_2 equals C_2's finishing time because the two tasks in stage three of C_2 are the last to complete.

With different combinations of tasks in a cohort, the Profiler can evaluate how tasks (from the same stage or different stages) compete for resources. With this information fed into the Solver, potential bottlenecks of the system can be identified, and task concurrency can be maximized.

4.2 The Solver

The Solver finds a *cohort set* K that consists of m cohorts: $K = \{C_i\}_{i=1..m}$. The cohort set represents a sequence of cohorts executed one after the other *ad infinitum*. For example, in Figure 3, the cohort set K consists of three cohorts, which are run repeatedly. The finishing time of the cohort set is $f(K) = \sum_{C \in K} f(C)$. We extend the notation for K to $K_{s_1,..,s_n}$ where s_i is the number of tasks at stage i across all the cohorts in K (see Figure 3 for a visual representation of this).

$K_{s_1,..,s_n}$ is *stable* if each stage s_i is servicing the same number of tasks j. A stable cohort set neither overloads nor starves any stage of tasks. For stable cohort sets, the job throughput is $tp = j/f(K_{s_1,..,s_n})$. Our first objective is to identify the stable cohort set $K^*_{s_1,..,s_n}$ with the best throughput. In order to find $K^*_{s_1=j,..,s_n=j}$, we construct a dynamic program which takes j as input.

Notice that in this scenario, $K^*_{s_1,..,s_n}$ can be split into subproblems $K^*_{t_1,..,t_n}$ and $K^*_{s_1-t_1,..,s_n-t_n}$ where $0 \leq t_i \leq s_i$. We construct an n-dimensional dynamic programming data structure with each entry computed as follows:

$$f(K^*_{s_1,..,s_n}) = \min_{t_1=0..s_1,...,t_n=0..s_n} \left[f(K^*_{t_1,..,t_n}) + f(K^*_{s_1-t_1,..,s_n-t_n}) \right]$$

Note that when $t_1 = t_2 = ... = t_n = 0$, the profiling-based measurements for a single cohort $C_{s_1,...,s_n}$ are used in place of $K_{s_1,...,s_n}$ on the right hand side in the equation above.

Throughput Maximization: The dynamic program starts from $j = 1$ and explores increasing values of j and has two natural terminating conditions which are quickly reached in practice: (1) either a schedule is found which satisfies 30fps, or (2) the solution of $j + 1$ yields no additional throughput gain over that of j, implying there is no further benefit of task parallelism. Formally, the complexity of computing a single data structure entry is j^n and the number of entries to be computed is j^n, resulting in a complexity of $O(j^{2n})$ for the dynamic program. The number of dataflow stages n is constant for the dataflow, so the optimization overhead is polynomial in j.

A convenient property of the Profiler and Solver is that job throughput is directly proportional to user frame rate. Given the job throughput tp above, the user frame rate is $tp_u = \frac{tp}{u}$ for u users. When new users join, tp_u is simply compared against the threshold 30fps to determine whether additional dynamic programming is necessary.

Latency Reduction: Now that we have identified the cohort set K^* with the best throughput, we aim to minimize the latency of jobs in K^*. For example, in Figure 3, Job 1 starts at C_1 and finishes in the next cohort set iteration at C_2. Note that the worst case latency is at most $n \times j$.[3] We reduce average latency by sorting the cohorts in K^* to produce a cohort sequence K^{**}. During runtime, the cohorts are then scheduled in succession according to this sequence.

Our sorting criteria are twofold: (a) cohorts that have tasks in earlier stages of the dataflow are scheduled earlier, and (b) cohorts that have fewer tasks in later stages of the dataflow are scheduled earlier. Based on these two criteria, a decimal-valued key $x.y$ can be constructed for each $C \in K^*$ where $x = i$ for the first non-zero value of s_i and $y = \sum_{j=i+1}^{n} s_j$. A sort over the keys yields the cohort sequence K^{**} that reduces latency significantly in practice. If the target λ_{host} ms latency is not reached, we continue exploring additional schedules with larger j.

Fairness: K^{**} is a cohort set with good throughput and latency, but it does not guarantee low variance in job latency. For example, if K^{**} consists of two jobs, the first job may finish in n cohorts whereas the second job may finish in $2n$ cohorts. Such extreme job latency differences can sometimes lead to persistent user-perceived response time unfairness when the number of users is the same as (or a multiple of) the number of jobs in K^{**}. Extending upon the above example, if there are two users, User Two will be consistently assigned to the same slow job. To mitigate this pathological case when the number of users u is a multiple of the number of jobs, we intentionally desynchronize job assignment with users by assigning each job in j to user id $u + 1$ if it was previously assigned to user id u. Desynchronized job assignment is both low cost and guaranteed to be fair in expectation.

User Churn: As mentioned above, when new users join, tp_u may already satisfy the throughput requirements, in which case K^{**} can continue to be used without issue (note that users leaving is not a problem). If it does not, the profiling and search for a new schedule is initiated while K^{**} continues to temporarily serve as the active schedule (albeit with substandard user experience). To generate the new schedule, it is only necessary to profile for larger cohorts starting at $j+1$ tasks per stage because existing profiling data up to j can be reused. Rerunning the Solver is very inexpensive, and is done whenever the incremental profile data is ready. If the Solver fails to find a satisfying schedule, the new user is evicted.

On the very first run (e.g. after the host user initially acquires the app), there is a delay to collect sufficient

[3]To see this, consider example $K = \langle\langle 0, 0, 1\rangle, \langle 0, 1, 0\rangle, \langle 1, 0, 0\rangle\rangle$.

profile information. We show in §8 that the overhead of new profiling from scratch is very low at under 30s, during which time users may still continue to play, albeit with suboptimal performance.

5. WIRELESS SCHEDULING

The wireless channel plays an important role in the interaction between the host and clients. All input from a client – whether continuous such as accelerometer or event-based such as multi-touch – must transit the wireless uplink, and the resulting video frame must transit the downlink before the player sees the effect of the input on her own screen. Late or lost input degrade user experience. Therefore, we wish to guarantee that input data's *transmission latency* is low. In addition, clients should respect deadlines, in the sense that input issued earlier by a first client should not be delayed due to contention with input issued later by a second client.

Satisfying these goals is challenging with standard 802.11n because a Prime host sends large volumes of downlink video traffic. Therefore, contention among clients and the host can incur large delays. As we will show in §8, the existence of downlink traffic significantly alters the delay and jitter of uplink traffic with regular 802.11. While 802.11e (QoS Mode) provides facilities for prioritized transmissions, §8 will show that it is in fact very unfair to early transmitters which are forced to backoff quickly upon contention.

802.11 Background: With the default CSMA setting, a client is required to sense the channel for $DIFS$ time after the last packet transmission before starting a count down timer. If the client senses that the channel is busy before the count down timer expires, then the timer is paused until the channel is free again and $DIFS$ time has passed. Upon expiration of the count down timer, the client sends its packet. If the transmission fails, a retry is initiated by restarting the above process. The count down timer is set as $n \times SlotTime$ where $SlotTime$ is a constant and n is a random number uniformly selected from the range $[0, CW]$ where CW represents the range of the contention window. CW starts off as $CWMin$ and doubles in size up to $CWMax$ whenever a retry occurs. In summary, the main parameters that impact the transmission latency are $DIFS$, $CWMin$ and $CWMax$. In standard 802.11, these parameters are fixed. In 802.11e, these parameters are fixed per each of seven predefined classes of traffic [15].

5.1 Protocol Design

The Wireless Scheduler enforces stochastic earliest deadline first job completion on a lossy channel among uncoordinated job submitters. The MAC design is shown in Figure 4 and is described next.

Low Latency: To provide low latency input, we start by prioritizing input traffic over downlink traffic. We alter the default contention mechanism to give uplink traffic a more aggressive contention window size. Specifically, input traffic uses lower $DIFS$ and CW values,

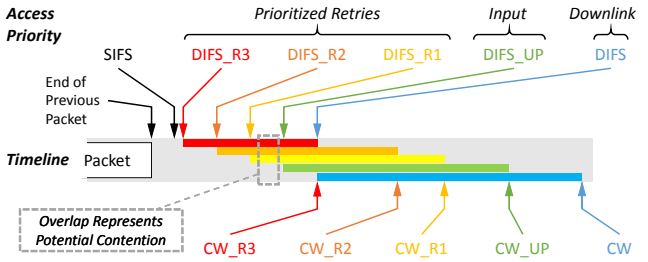


Fig. 4. Wireless Scheduler's Low Latency 802.11-compatible MAC. Input and its retries receive priority over video downlink traffic. Colored bars represent the time window of opportunity to acquire the channel.

$DIFS\_UP$ and CW_UP. As shown in Figure 4, input traffic wait time is probabilistically lower than downlink wait time, and therefore has a greater chance of acquiring the channel. Prioritizing uplink traffic has negligible impact on downlink performance simply because uplink traffic is low volume compared to downlink traffic, which we will show in §8. Also, downlink traffic is naturally resilient to late or lost packets by virtue of the fact that H.264 video has built-in loss compensation.

Fairness to Avoid Priority Inversion: Even though input streams are low volume and prioritized over downlink streams, input packet delivery may still fail due to channel fading and external interference. Default packet retransmission can provide reliability, but exponential backoff not only quickly increases total transmission latency, it creates opportunities for deadline priority inversion. Consider a first client's transmission of input packet p_1, which fails on the first attempt. If the retry of p_1 must compete for the channel with a new input packet p_2 from a second client, the retry will have a lower probability of acquiring the channel due to exponential backoff. In this case, p_1 has suffered priority inversion since it was ready for transmission earlier than p_2 but is now ordered after p_2. In fact, the greater number of times a packet is retransmitted, the worse its latency, which is exactly counter to our design goal where we are attempting to provide low transmission latency. Priority inversion afflicts both standard 802.11 and 802.11e since retransmitted packets inherit the priority of the initial transmission. The severity of priority inversion increases when clients are synchronized in their desire to transmit input, which is a naturally occurring phenomenon in competitive MDAs (such as users competing to answer a quiz question first). Therefore, despite a low average input bitrate, clustering of input transmissions causes significant priority inversion.

To solve priority inversion, input packets employ *priority escalation:* the earlier a packet is created, the higher the priority it is assigned during channel contention. We accomplish this by changing the priority of retransmitted packets. We design three additional priority levels for retransmitted packets. Priorities are defined by $DIFS$ and CW levels, where: $CW\_R1 > CW\_R2 > CW\_R3$ and $DIFS\_R1 > DIFS\_R2 > DIFS\_R3$. As shown

Priority Levels	$AIFSN^{\dagger}$	$CWMin$	$CWMax$
Output Video‡	3	15	1023
Input Initial	2	7	15
Input Retry #1	1	7	15
Input Retry #2	1	3	7
Input Retry #3	0	1	3

† $DIFS = AIFSN \times SlotTime + SIFS$

‡ Downlink settings are the same as standard 802.11n.

Table 2. Prime MAC configuration parameters.

in Figure 4, this mechanism probabilistically guarantees that packets created earlier acquire the channel.

Fairness Despite Link Quality Variance: An additional advantage of priority escalation is that it provides fairness among clients by mitigating the impact of imbalances in clients' link quality to the host. Even in the same room, clients may have significantly varying link qualities to a host or AP, as we found when looking at empirical wireless data traces [35]. Clients with lower link qualities experience higher packet error rates and undergo more retransmissions. By prioritizing retransmissions, Prime ensures that clients with high packet error rates are not unduly penalized, thereby providing an additional measure of fairness among clients.

802.11 Compatibility: Prime's modified MAC not only co-exists with existing 802.11 traffic, it can be built with existing 802.11 device support so that existing clients can easily adopt it. Specifically, we repurpose NIC components specified by 802.11e which exposes seven queues, each of which can have separate $DIFS$ and CW parameters. We modify this existing functionality to implement our five priority levels. Control logic in the wireless driver decides the priority for each packet by placing it into the appropriate queue. Whenever a client input packet is needing retransmission, it is moved to the next higher priority queue. The parameter settings for each queue are shown in Table 2. Note that the minimum setting for our $DIFS$ is $SIFS$, which is the minimum wait time that allows for acknowledgments. The embodiment of these 802.11 modifications is a modified Wi-Fi driver which can be installed as part of the one time Prime client software installation.

6. SCALABLE VIDEO ENCODING

Our early investigations revealed that despite multi-core CPUs on devices, encoding was a significant barrier to scalability. This section discusses the techniques Prime uses to alleviate encoding bottlenecks. We first review existing encoding methods and highlight their deficiencies which are exacerbated when a single host supports multiple clients. Then, we review codec accelerators and describe how the Prime Encoder uses them as building blocks for relieving encoding overhead.

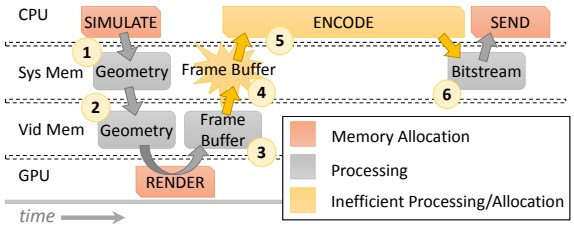
CPU Encoding: Figure 5(a) shows the processing and data transfer operations when CPU-based software encoding is used [34]. The app first generates geometry (refer to ① in Figure 5(a)), queues these for graphics processing ②, which the GPU uses to generate a frame ③. The frame buffer is copied back to system memory ④ for the CPU to initiate frame encoding ⑤. The encoded result is a bitstream that is transmitted to the client. Two main inefficiencies exist in this design. First, *memory copy* overhead from frame buffers ④ is problematic because the data volume is substantial compared to the memory bus bandwidth. Each frame is transferred uncompressed because of the GPU's data format requirements, and transfers happen continuously as fast as frames can be rendered (nominally 30fps). Second, *CPU saturation* occurs because encoding on the CPU is slow (even with media-targeted ISAs such as AVX), and blocks other work such as application state update.

Prime Encoding: The ubiquity of cameras on devices has led to mainstream popularity of recording mobile video clips. In response, all major SoC vendors including Qualcomm, Samsung, NVIDIA and Intel have introduced dedicated encoding logic to address the common case of compressing mobile video clips (e.g., for local storage or email transmission). Prime re-purposes these codec accelerators to greatly reduce the overhead of encoding of multiple streams rendered in real-time. As an illustrative example, Intel places dedicated encoding silicon alongside existing GPU and CPU cores. The dedicated encoder handles only the serial stages of the encode process (entropy encoding) while the GPU handles the parallelizable encoding operations.

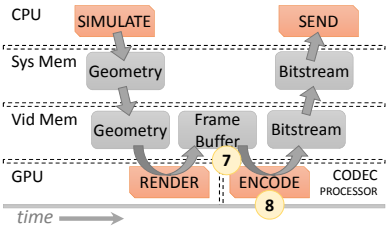
The Prime Encoder starts by replacing CPU encoding with dedicated encoding logic, as shown in ⑧ of Figure 5(b). Next, since the input to the encoder is a rendered frame buffer already in GPU memory, we take advantage of this opportunity to bypass expensive frame buffer memory copying as incurred in ④. Rather, we modified the codec accelerator data structures to directly accept and frame buffers in video memory as input as shown in ⑦. Afterward, the accelerator transfers the final, compressed bitstream to system memory. Finally, the CPU only performs a lightweight network send, which eliminates encoding overhead on the CPU. These changes address the limitations of standard software encoders. Note that contention may still occur between encoding and rendering since both utilize GPU cores; the Host Scheduler manages this implicit contention.

7. PROTOTYPE IMPLEMENTATION

The Prime prototype is implemented as a dataflow application framework. The Prime framework is 5,100 loc written in C++ for Windows. The scalable encoder is 2,300 loc and interfaces with Intel's family of on-chip codec accelerators [16]. Additionally, the codec accelerator also performs transcoding to upsample or downsample from the host resolution to the client resolution should they differ. The MAC protocol is implemented as a modification of the wireless NIC driver, with the required changes touching 70 loc. The wireless bandwidth is monitored in realtime using packet pairs [43], and each client stream's bitrate is set to $\frac{1}{n}$ of the bandwidth. Clients communicate over Wi-Fi Direct initiated



(a) Software Encoder



(b) Prime Encoder

Fig. 5. Encoder Pipeline Comparison

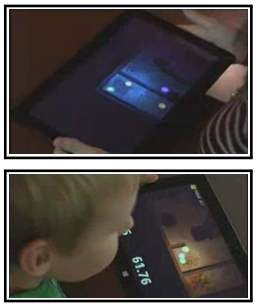


Fig. 6. Users playing Marble-Maze with Prime

by NFC tap using a VNC-like client terminal app that is 5,300 loc. We next describe several additional salient elements of the implementation.

Workload. As an example workload, we have taken MarbleMaze, a production 3D game for ARM tablets [27], and ported it to run on the Prime framework. In the game, the player's objective is to guide a marble (via touch and accelerometer-based tilt) through a labyrinth without succumbing to any of the various snares en route (see Figure 6 for shots of engaged players); players care about responsiveness because fast reaction times determine who wins. In its original form as a single player game, MarbleMaze is 9,200 lines of code (loc). We modified it to support head-to-head simultaneous multiplayer, which increased its complexity to 12,000 loc.

Dataflow API. MarbleMaze implements Prime's dataflow by exposing view-independent and view-dependent dataflow components that are responsible for handling a limited set of upcalls: `viewIndependentReady` indicates that new input is available and that the application should start processing a new task for the view independent stage. Similarly, `viewDependentReady` indicates that player-specific task processing is ready to start. `viewIndependentDone` and `viewDependentDone` are the matching downcalls to indicate corresponding completion events. Lastly `clientJoin` is triggered when a new client joins and app-specific state initialization should be undertaken. While we have yet to port other games to this API, dataflow maps particularly well to games where the vast majority are already built on game engines that structure user code according to dataflow-like stages of input processing, simulation and rendering [11, 39].

8. EVALUATION

The objective of our evaluation is to answer the following question: how many users can Prime scale to while maintaining high frame rate, responsiveness and fairness? A summary of our findings is as follows.

- The Host Scheduler finds schedules that can support frame rates that are from $1.4 - 2.3\times$ more than with naïve schedules, with a median latency of 22ms and good fairness.
- The Wireless Scheduler both reduces input latency and degrades downlink throughput less than standard 802.11n and 802.11e, especially during periods of high activity when input traffic is abundant.
- The Encoder Pipeline increases encoding throughput by $5.5 - 14\times$ over software encoding, eliminating encoding as an obvious bottleneck.
- The end-to-end touch-to-pixel latency is below 100ms, with a median of 54ms and 99%-tile of 94ms.

Methodology: The metrics for assessing successful scalability are (1) throughput as measured by per user frame rate, (2) latency as measured by the time required for generating a single frame which consists of both wireless latency and host device processing latency, and (3) fairness as measured by variance in latency across users. As is standard for gaming scenarios, we assume that all devices are only running the immersive MDA in the foreground. Performance of background apps is not a central concern.

Our host device is a Windows Surface Pro tablet with Intel Core i5, 4GB memory, an integrated Intel HD Graphics 4000 GPU, and 1920×1080 pixel display. Our client devices are Samsung Series 7 Slate tablets running Windows with Intel Core i5 CPU, 4GB memory, integrated Intel HD Graphics 3000 GPU, 1366×768 pixels. We use these host and client devices to evaluate encoding performance and the Host Scheduler. To evaluate the Wireless Scheduler, we use Linux because the Linux Qualcomm ath9k wireless drivers are open source. Our wireless performance evaluation uses the TP-LINK TL-WDN4800 802.11n NIC hardware. We perform our experiments by collecting user input traces, and replaying these on the production system for repeatability.

Host Scheduling Performance: We compared Host Scheduler against two baselines. *Baseline-1* a naïve scheduler which issues one task to every stage at regular intervals, cycling through clients in a round-robin fashion (or equivalently, identifying the bottleneck stage and scheduling tasks at the rate of task completion at the bottleneck). *Baseline-N* is also similar to the naïve schedule except stages may process up to n tasks concurrently, where n is the number of users. Baseline-1 is conservative in that it exploits no intra-stage parallelism whereas Baseline-N is aggressive in that it assumes every stage is capable of n-way intra-stage parallelism. Both permit inter-stage parallelism by concurrently schedul-

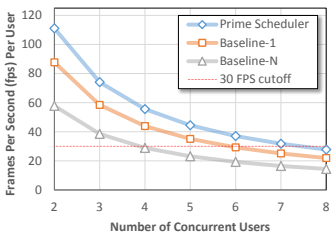


Fig. 7. User Framerate w/ Scheduling

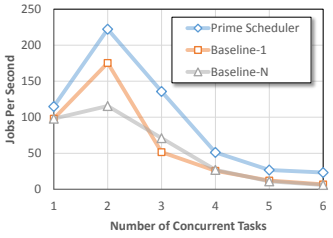


Fig. 8. Scheduler Job Throughput

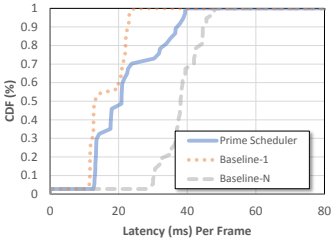


Fig. 9. Frame Generation Latency

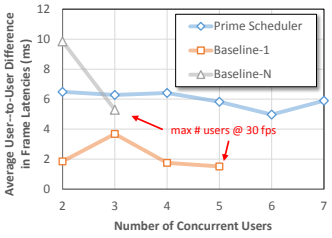


Fig. 10. Fairness Across Users

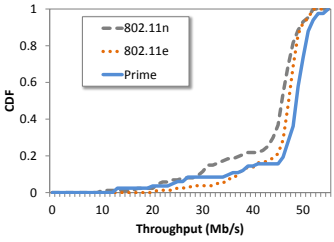


Fig. 11. Input TX Latency

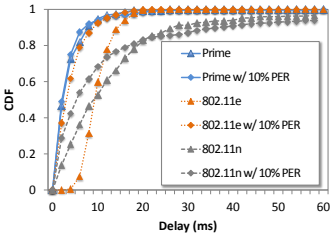


Fig. 12. TX Latency with PER

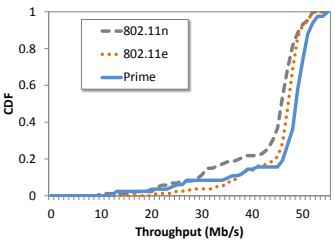


Fig. 13. Downlink Throughput

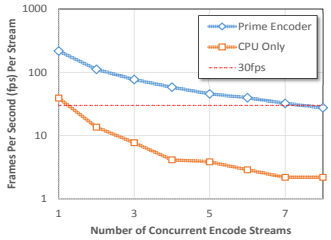


Fig. 14. Encoding Throughput

ing work across stages. Note that our early prototype already showed the shortcomings of best-effort schedulers (§2).

Figure 7 shows each scheduler's frame rate per player as the number of players increases. Prime Scheduler is able to support 7 players above 32fps whereas Baseline-N and Baseline-1 can only support 5 and 3 respectively, an improvement of 1.4× and 2.3×. Note that these measurements are with the improved Encoder Pipeline installed. Taking a closer look at the underlying schedulers' performance, Figure 8 shows the job throughput of Prime Scheduler versus baselines. Recall one job corresponds to one instance of converting user input to a frame. The benefits of parallelism peak at two concurrent tasks. Prime Scheduler is much better able to take advantage of parallelism than either Baseline-1 or Baseline-N. Figure 9 shows responsiveness in terms of the time required to generate one frame. Prime Scheduler shows responsiveness with a median latency of 21ms as compared to Baseline-1 and Baseline-N with 13ms and 38ms, respectively. Even though Prime supports more simultaneous players, its latency is comparable to baseline schemes. Prime Scheduler delivers fairness across users as well. Figure 10 shows the difference in latencies between each pair of users, averaged across all pairs. The difference is calculated as the Euclidean distance. As the number of users increases from two to seven, Prime Scheduler is able to maintain all users with latencies within 6.5ms of each other. Baseline schemes provide similar or better latencies, but scale to fewer number of users.

The overhead of Prime Scheduler is its need to perform profiling and run the dynamic program scheduler. Initial profiling takes 22.5 seconds on the host device, which is a very modest one time cost and *not* incurred whenever a game is started nor when a player joins (see §4). After initial profiles are collected, maintaining profiling data is a negligible runtime cost. The cost to run the dynamic program is also negligible.

Wireless Scheduling Performance: We evaluate Wireless Scheduler and find that it delivers lower latency and better fairness with the same reliability and less degradation on downlink throughput compared to standard 802.11n and 802.11e. Input consists of an accelerometer sampled at 8Hz continuously, and touch which is sampled at 8Hz upon activation of a touch event. We first note two operating regimes. When there is little input traffic, e.g. when players are in a quiescent part of the game, all three protocols perform similarly. Conversely, when there is substantial input traffic, e.g. when players are in an active part of the game and aggressively submitting input, the advantages of Wireless Scheduler are well articulated. Figure 11 shows the input transmission latency measured during several such periods of high activity with six clients. First, we see that regular 802.11n performs extremely poorly, with 5% of latencies exceeding 26ms and 1% of latencies over 110ms. Second, while 802.11e performs better than standard 802.11n, 5% still exceed 10ms and 1% exceed 90ms. The implication is that more than a dozen input readings are delayed by tens of milliseconds every second, which is a delay that is cumulative with host processing latency. In contrast, our Wireless Scheduler's escalated priorities for retries mitigates priority inversion, resulting in short input transmission latencies with 95% finishing in under 3ms and 99% finishing within 14ms.

Next, we look at fairness among clients when clients may have different link qualities to the host. Figure 12 shows the impact when one client out of six has a 10% Packet Error Rate (PER). Despite the PER, all six Prime clients have similar input transmission latencies with a 80th percentile of 4ms. However, with 802.11e, five other clients' transmission latencies suffer with a 80th percentile trans-

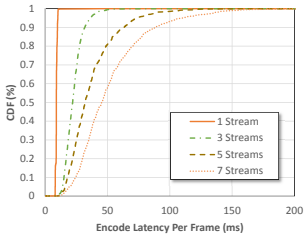


Fig. 15. Encoding Latency w/o Scheduling

Touch-to-Pixel	
%-tile	Latency
50th	53ms
95th	78ms
99th	94ms

Fig. 16. End to End Touch-to-Pixel Latency.

mission latency of 12ms. Standard 802.11n is even worse with high PER at 19ms.

Lastly, our Wireless Scheduler impacts downlink traffic less than 802.11e and 802.11n. Figure 13 shows that Prime median downlink throughput is 49Mbps versus 47Mbps and 45Mbps for 802.11e and 802.11n respectively. This result can be explained by the fact that Prime tends to quickly acquire the channel and finish transmitting, avoiding the back and forth rounds of collisions that can plague protocols whose input traffic competes with downlink traffic.

Encoding Performance: CPU-based encoding severely limits frame throughput and is easily identified as the first bottleneck to scalability. Fortunately, the Encoder Pipeline increases frame throughput by $5.5 - 14\times$. Figure 14 shows the large frame rate improvement on input of 1920×1080. Software encoding of just one stream barely achieves 30fps with CPU utilization at 100%, retarding any other useful work. In contrast, our encoding pipeline can simultaneously service seven clients at 32fps while lightly loading the CPU.

In addition, Figure 15 shows that our Encoder Pipeline also delivers a median response latency of 32ms for five clients and 45ms for seven clients. However, the response latency becomes highly variable as the number of clients increases, with very large maximum response times possible. The Host Scheduler's profiling observes this variance and accommodates it in its scheduling decisions.

Energy Overhead: The host device spends additional energy to host client instances. We measured the host device power draw with a WattsUp power meter. Starting from a baseline of 16.9 Watts for the host device to service a single player (itself), the additional power required to host seven players is 3.2 Watts. This reduces the host's 42Wh battery by an acceptable 4% over the course of a 30 minute gaming session.

End-to-End Latency: Lastly, we evaluate the overall touch-to-pixel latency λ_{e2e}. We evaluate this by combining results of our wireless testbed and host scheduler. As shown in Table 16, our final system can achieve a median latency of 54ms and 99%-tile of 94ms, which is well below the latency budget of 100ms [3, 10]. This ensures that all seven users' interactions with the MDA will be smooth and responsive.

9. RELATED WORK

One longstanding theme of ubiquitous computing has been to enable physically co-located users to participate in the same digital setting. One thread of this work is that on large shared displays where researchers have sought to enable better impromptu collaborations and effective contextualization of digital artifacts [36, 37, 14, 42, 41]. The popularity of mobile devices has meant that for small groups, it is usually a collection of moderate displays rather than a single large display which is often the most readily available. In this vein, several works have prototyped systems infrastructure for MDAs [29, 28]. Unlike Prime, these early works do not tackle platform heterogeneity, nor do they address quality metrics such as frame rate, response latency and fairness.

Much work has looked at using mobile devices as VNC clients [22, 8, 23, 9, 47, 38, 18]. Our work is distinct in proposing devices as hosts. The overall MDA scheduling problem across host and clients belongs to the class of "flexible flow shop" problems in which a set of jobs must transit through a sequence of fixed processing stages, each of which may have multiple processors available. Common optimization objectives are either throughput or latency (see [30, 31] for surveys). Our work addresses a more challenging problem than this prior work because it must satisfy both throughput and latency requirements, in addition to fairness. Also as discussed extensively in §4, the idealized flow shop model is inaccurate because simple assumptions about concurrency and contention between stages are grossly misguided in practice.

The Wireless Scheduler's multiple objectives of providing low latency, reliability, fairness across link qualities, and compatibility with 802.11 is a novel design point in the wireless community. Several wireless MACs are targeted at low latency alone [17, 33, 24, 12]. A number of 2.4Ghz TDMA approaches exist that address latency and reliability [19, 32], but they are not backward-compatible with 802.11. With regards to 802.11-compatible MAC modifications, SoftSpeak [40] addresses latency of wireless channel acquisition for IP telephony. It assumes reliability without retransmissions, and hence is best suited for data streams which already contain loss compensation, such as encoded audio.

10. CONCLUSION

As mobile devices proliferate, users will naturally imagine interacting not just with one device at a time, but with a host of nearby devices – both their own and those of other users. With a simple remote display abstraction, multi-device apps can instantly span an ad hoc set of nearby devices. Prime's improvements to real-time encoding, wireless latency, and host resource scheduling enable as many as seven players to participate simultaneously while maintaining reasonable frame rate, low response latency and fairness for all users.

11. REFERENCES

1. R. A. Baratto, L. N. Kim, and J. Nieh. Thinc: a virtual display architecture for thin-client computing. *SIGOPS Operating System Review*, 39(5):277–290, Oct. 2005.
2. L. Barkhuus, M. Chalmers, P. Tennent, M. Hall, M. Bell, S. Sherwood, and B. Brown. Picking pockets on the lawn: The development of tactics and strategies in a mobile game. In *Proc. of the 7th International Conference on Ubiquitous Computing*, UbiComp'05, pages 358–374, Berlin, Heidelberg, 2005. Springer-Verlag.
3. 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 *Proc. of NetGames*, 2004.
4. M. Bell, M. Chalmers, L. Barkhuus, M. Hall, S. Sherwood, P. Tennent, B. Brown, D. Rowland, S. Benford, M. Capra, and A. Hampshire. Interweaving mobile games with everyday life. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 417–426, New York, NY, USA, 2006. ACM.
5. S. Bjrk, J. Falk, R. Hansson, and P. Ljungstrand. Pirates! using the physical world as a game board. In *Proc. of Interact 2001*, pages 9–13, 2001.
6. L. Brunnberg and O. Juhlin. Keep your eyes on the road and your finger on the trigger - designing for mixed focus of attention in a mobile game for brief encounters. In *Proc. of the 4th International Conference on Pervasive Computing*, PERVASIVE'06, pages 169–186, Berlin, Heidelberg, 2006. Springer-Verlag.
7. J. Clawson, A. Voida, N. Patel, and K. Lyons. Mobiphos: A collocated-synchronous mobile photo sharing application. In *Proc. of the 10th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '08, pages 187–195, New York, NY, USA, 2008. ACM.
8. E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 121–135, New York, NY, USA, 2015. ACM.
9. D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester. A hybrid thin-client protocol for multimedia streaming and interactive gaming applications. In *Proc. of the 2006 international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '06, pages 15:1–15:6, New York, NY, USA, 2006. ACM.
10. M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *Proc. of NetGames*, 2005.
11. Epic Games. Unreal engine. `http://www.unrealengine.com`.
12. E. Felemban, C.-G. Lee, and E. Ekici. Mmspeed: multipath multi-speed protocol for qos guarantee of reliability and. timeliness in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 5(6):738–754, June 2006.
13. 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`.
14. E. M. Huang and E. D. Mynatt. Semi-public displays for small, co-located groups. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, pages 49–56, New York, NY, USA, 2003. ACM.
15. IEEE. Part 11 amendment 8: Mac qos enhancements. `http://standards.ieee.org/getieee802/download/802.11e-2005.pdf`.
16. Intel Corp. QuickSync Programmable Video Processor. `http://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html`.
17. K. Jamieson, H. Balakrishnan, and Y. C. Tay. Sift: A mac protocol for event-driven wireless sensor networks. In *Proc. of EWSN*, 2006.
18. J. Kim, R. A. Baratto, and J. Nieh. An application streaming service for mobile handheld devices. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 0:323–326, 2006.
19. R. R. Kompella, S. Ramabhadran, I. Ramani, and A. C. Snoeren. Cooperative packet scheduling via pipelining in 802.11 wireless networks. In *Proc. of SIGCOMM E-WIND*, 2005.
20. A. Lai and J. Nieh. Limits of wide-area thin-client computing. In *Proc. of SIGMETRICS*, 2002.
21. J. R. Lange, P. A. Dinda, and S. Rossoff. Experiences with client-based speculative remote display. In *Proc. of ATC*, 2008.
22. K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 151–165, New York, NY, USA, 2015. ACM.

23. S. F. Li, Q. Stafford-Fraser, and A. Hopper. Integrating synchronous and asynchronous collaboration with virtual network computing. *IEEE Internet Computing*, 4(3):26–33, May 2000.
24. G. Lu, B. Krishnamachari, and C. Raghavendra. An adaptive energy-efficient and low-latency mac for data gathering in wireless sensor networks. In *Proc. of 18th International Parallel and Distributed Processing Symposium, 2004.*, April 2004.
25. A. Lucero, J. Holopainen, and T. Jokela. Pass-them-around: Collaborative use of mobile phones for photo sharing. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1787–1796, New York, NY, USA, 2011. ACM.
26. Microsoft. Microsoft Remote Desktop Protocol. `http://msdn.microsoft.com/en-us/library/aa383015.aspx`.
27. Microsoft. Developing marblemaze, a windows store game. `http://msdn.microsoft.com/en-us/library/windows/apps/br230257%28v=vs.110%29.aspx`, 2011.
28. T. Pering, K. Lyons, R. Want, M. Murphy-Hoye, M. Baloga, P. Noll, J. Branc, and N. De Benoist. What do you bring to the table?: Investigations of a collaborative workspace. In *Proc. of the 12th ACM International Conference on Ubiquitous Computing*, UbiComp '10, pages 183–192, New York, NY, USA, 2010. ACM.
29. T. Pering, R. Want, B. Rosario, S. Sud, and K. Lyons. Enabling pervasive collaboration with platform composition. In *Proc. of the 7th International Conference on Pervasive Computing*, Pervasive '09, pages 184–201, Berlin, Heidelberg, 2009. Springer-Verlag.
30. M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 3rd edition, 2008.
31. K. Pruhs, E. Torng, and J. Sgall. Online scheduling. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, 2004.
32. A. Rao and I. Stoica. An overlay mac layer for 802.11 networks. In *Proc. of MobiSys*, 2005.
33. I. Rhee, A. Warrier, M. Aia, and J. Min. Z-mac: a hybrid mac for wireless sensor networks. In *Proc. of SenSys*, 2005.
34. T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *Internet Computing*, 2(1):33–38, 1998.
35. M. Rodrig, C. Reis, R. Mahajan, D. Wetherall, and J. Zahorjan. Measurement-based characterization of 802.11 in a hotspot setting. In *Proc. of SIGCOMM E-WIND*, 2005.
36. D. M. Russell, C. Drews, and A. Sue. Social aspects of using large public interactive displays for collaboration. In *Proc. of the 4th International Conference on Ubiquitous Computing*, UbiComp '02, pages 229–236, London, UK, UK, 2002. Springer-Verlag.
37. C. Shen, K. Everitt, and K. Ryall. Ubitable: Impromptu face-to-face collaboration on horizontal interactive surfaces. In *In Proc. UbiComp 2003*, pages 281–288, 2003.
38. S. Shi, C.-H. Hsu, K. Nahrstedt, and R. Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proc. of MM*, 2011.
39. Unity Technologies. Unity Game Engine. `http://unity3d.com/`.
40. P. Verkaik, Y. Agarwal, R. Gupta, and A. C. Snoeren. Softspeak: making voip play well in existing 802.11 deployments. In *Proc. of NSDI*, 2009.
41. D. Vogel and R. Balakrishnan. Interactive public ambient displays: Transitioning from implicit to explicit, public to personal, interaction with multiple users. In *Proc. of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, pages 137–146, New York, NY, USA, 2004. ACM.
42. D. Wigdor, H. Jiang, C. Forlines, M. Borkin, and C. Shen. Wespace: the design development and deployment of a walk-up and share multi-surface visual collaboration system. In *Proc. of SIGCHI*, 2009.
43. Q. Xu, S. Mehrotra, Z. Mao, and J. Li. Proteus: network performance forecast for real-time, interactive mobile applications. In *Proc. of MobiSys*, 2013.
44. S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari. The performance of remote display mechanisms for thin-client computing. In *Proc. of ATEC*, 2002.
45. N. Zhang, Y. Lee, M. Radhakrishnan, and R. Balan. Gameon: p2p gaming on public transport. In *MobiSys 2015*, 2015.
46. Y. Zhou, G. Percival, X. Wang, Y. Wang, and S. Zhao. Mogclass: Evaluation of a collaborative system of mobile devices for classroom music education of young children. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 523–532, New York, NY, USA, 2011. ACM.
47. M. Zhu, S. Mondet, G. Morin, W. T. Ooi, and W. Cheng. Towards peer-assisted rendering in networked virtual environments. In *Proc. of MM*, 2011.