DOHA: Scalable Real-time Web Applications through Adaptive Concurrent Execution

Aiman Erbad
University of British Columbia
201-2366 Main Mall
Vancouver, BC, Canada
aerbad@cs.ubc.ca

Norman C. Hutchinson
University of British Columbia
201-2366 Main Mall
Vancouver, BC, Canada
norm@cs.ubc.ca

Charles 'Buck' Krasic
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA, USA
ckrasic@google.com

ABSTRACT

Browsers have become mature execution platforms enabling web applications to rival their desktop counterparts. An important class of such applications is interactive multimedia: games, animations, and interactive visualizations. Unlike many early web applications, these applications are latency sensitive and processing (CPU and graphics) intensive. When demands exceed available resources, application quality (e.g., frame rate) diminishes because it is hard to balance timeliness and utilization. The quality of ambitious web applications is also limited by single-threaded execution prevalent in the Web. Applications need to scale their quality, and thereby scale processing load, based on the resources that are available. We refer to this as *scalable quality.*

DOHA is an execution layer written entirely in JavaScript to enable scalable quality in web applications. DOHA favors important computations with more influence over quality based on hints from application-specific adaptation policies. To utilize widely available multi-core resources, DOHA augments HTML5 web workers with mechanisms to facilitate state management and load-balancing. We evaluate DOHA with an award-winning web-based game. When resources are limited, the modified game has better timing and overall quality. More importantly, quality scales linearly with a small number of cores and the game is playable in challenging scenarios that are beyond the scope of the original game.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Design, Performance

Keywords

Quality of Service, HTML5 games, Web workers, JavaScript

1. INTRODUCTION

The Web has evolved from a distributed document repository to become the de-facto distributed application platform. As a platform its success is unparallelled. It supports an enormous variety of complex and rich applications and services, with the functionality of web applications often rivalling that of their desktop counterparts. The trend of adopting the Web platform is expected to continue because web applications have faster maintenance and deployment cycles as well as good portability. As ambitious applications migrate to the Web, they need to maintain rich user interfaces and support interactive scenarios making performance a major concern in web-clients. Until recently, interactive multimedia applications were limited by the lack of key technologies, such as rich graphics elements, bi-directional continuous network transport, and fast JavaScript engines. Browser capabilities and the tools available now allow building complex web applications, such as games using only HTML, CSS, and JavaScript. Modern browsers are improving in terms of performance and features at an exceptional pace but web applications are becoming more ambitious and resource intensive.

Web-based games are great representatives of ambitious (media-rich and time-sensitive) applications which are moving to the Web. Games and game developers are at the edge of technology, often pushing the boundaries of what is possible. Similar to desktop games [4], popular web games [8, 24, 25] use most of the available processing power (between 80%-100% of a 2GHz core). Games and other ambitious web applications are shifting the performance optimization focus from the download and parsing time of web files to the run-time performance. According to the developers [8, 25], adding new features is limited by the available processing capability. Developers spend significant effort optimizing after adding each feature and are forced to remove or simplify some features to match the available capability. This problem is exacerbated if developers target all execution platform combinations of browsers/browser versions (Chrome, Firefox 4, Firefox 6, etc), operating systems (Windows, Mac, Android and Linux), and hardware processing capability (mobile to high-end PC).

Many web applications have adopted an event-based programming model [22] in order to be responsive. When the demand exceeds available CPU resources, however, it is not feasible to execute all application events (callback functions) in a timely fashion. The browser best-effort execution model does not provide any mechanism to balance between timeliness and utilization. To port interactive web applications, developers hardcode the appropriate settings, such as the game target frames per second. This approach cannot keep up with the expanding number of platform combinations. More importantly, it does not gracefully handle the dynamic fluctuations in application demands (common in multimedia applications) or available resources (due to sharing the CPU with other applications) over time. Without a general solution that scales demand to available resources, the perceived quality of these applications becomes brittle and sensitive to any change in the execution conditions.

The quality of ambitious applications is also limited by single-threaded execution prevalent in web browsers. These applications need more processing power than available in one core especially in mobile platforms with low-end cores. Multi-core processors are now available in most computing platforms (desktops, laptops, tablets, and smart phones) since hardware trends favor parallel architectures. To deliver the available CPU cycles to web applications, we need to facilitate concurrent software development. HTML5 web workers [10] introduce a shared-nothing concurrency model as the first step toward concurrent web applications, as we see in Figure 1. Although this enables concurrent execution, it does not help developers address challenging concurrency issues, such as state management and load-balancing.

DOHA is an execution layer on top of JavaScript engines that enhances the event-driven concurrency model. With hints from applications, DOHA guarantees timely dispatch for important events and scales application demands with available resources. DOHA reduces the challenges of developing concurrent web applications unleashing the potential of widely available multi-core processors. To be more specific, we make the following contributions:

- We define scalable quality as a necessary requirement to write web applications once and run them with consistent quality everywhere. Scalable quality ensures applications degrade quality gracefully when demands exceed resources and scale quality up when more resources are available.

- We designed and implemented an execution layer which includes HTML5 web workers to enable scalable quality in interactive multimedia web applications. To utilize multi-core resources, DOHA augments HTML5 web workers with mechanisms to ease handling challenging concurrency issues, such as state management and load-balancing.

- While re-structuring the simulation engine of an award-winning web-based game (RAPT [25]), we examine the challenges and opportunities of using HTML5 web workers and share our qualitative and quantitative observations.

The modified game (with DOHA) has better timing and higher perceived quality when resources are scarce. More importantly, the overall quality scales linearly as we use more cores (up to 3 cores in RAPT). Our parallel game is playable in larger game scenarios beyond the scope of the original game. The remainder of the paper is structured as follows: Section 2 discusses DOHA's design and implementation details, Section 3 presents our evaluation results, Section 4 describes our qualitative lessons learned, Section 5 highlights related work, and Section 6 concludes.

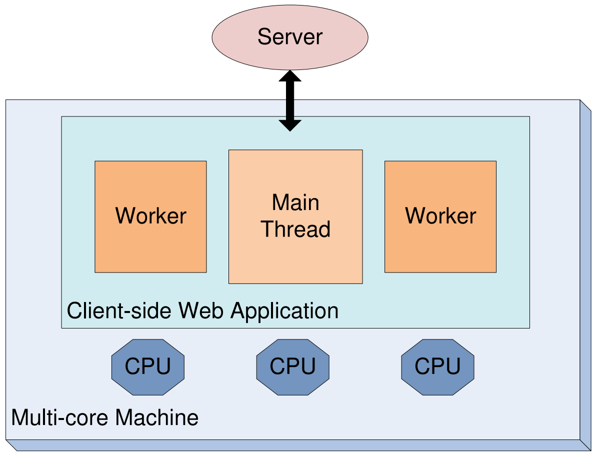

Figure 1: Web application with two HTML5 workers running in a multi-core platform

```
function update (time) {
  // Call update for all entities
  for(entity in game_entities){
      entity.update (time);
  }
}
```

Figure 2: Game loop global update function

2. DESIGN AND IMPLEMENTATION

While studying the architecture of various web-based games [1, 8, 24, 25], we observed that they have one or multiple execution loops for the basic tasks, such as rendering and simulation. As we see in Figure 2, current game loops have a global update that iterates over all game entities (e.g., players and enemies) in a pre-determined order (creation time order in RAPT [25] and z-axis ordering in the Render Engine [8]). The global update is called using a JavaScript timer 30 to 60 times a second depending on the target frame rate. Current game loops attempt to update all entities at each frame in a timely fashion. This architecture leads to brittle application quality because adding one feature affects frame duration and can render the game not playable.

DOHA provides web applications with abstractions and an execution layer to have better control over quality and enable access to available multi-core resources. DOHA consists of two major components: the event-loop which handles prioritized execution locally in each thread, and the concurrent execution module, MultiProc, which simplifies state management and scheduling of events on worker threads.

2.1 Event-loop

An event-based architecture is a natural fit for the asynchronous browser execution environment. DOHA's event-driven programming model is inspired by the principles of reactive programming [3] and aims to support the needs of interactive web applications. Popular web applications are event-driven with a large number of short callback functions [22]. DOHA introduces explicit execution events that specify the function to be executed and the call parameters. Events in DOHA give the underlying scheduler performance hints. Events also define the granularity at which applications adapt (scale quality up and down). Inspired by priority-progress adaptation [15], elastic computationally intensive parts of games are broken into explicit events.

Our key observation is that time-sensitive applications have some computations that are time synchronous (e.g., sound and game loop updates) and others that are best-effort (e.g., AI logic and the particle engine) and can be adapted. These two types of computations need to be clearly identified so that their needs can be met independently. The event-loop dispatches non-preemptively, prioritizing the time synchronous computations over the best-effort computations [14]. Events can be dispatched with low latency because our event-based model should ideally have short-lived computations that avoid blocking.

The key primitives in the event-loop are: *submit* an event for execution (and start the execution loop if it was not active), *run* to start the execution loop, *cancel* to delete a submitted event before it is executed, and *stop* to pause the execution loop. Each event is given a type, a callback specifying the function that will be called, and an array of arguments. Explicit execution events have two types: *timer and best-effort*. In timer events, the release time specifies the time an event becomes eligible to execute. Once eligible, timer events take precedence over best-effort events. For best-effort events, execution is ordered according to priority. When application demands exceed available resources, it is not feasible to dispatch all events in a timely fashion. Best-effort events with more influence over perceived quality are given high priority so they execute first. Less important events are cancelled when they become stale (based on time) matching demands to available resources. Priority and release time are assigned by an application-specific adaptation policy.

To order the execution of both event types, the event-loop has two internal priority queues. Timer events have a min-heap so events with earlier release times are closer to the heap root. Best effort events have a max-heap so higher priority events are closer to the root. At each event-loop iteration, we peek at the timers heap root to examine the closest release-time. If it has been reached, we execute the root event. If it has not been reached, we execute the best-effort heap root. If the best-effort events heap is empty, we yield to the underlying JavaScript engine until the closest release-time. If the timers heap is empty, we yield execution of the event-loop until a new event is submitted.

When an event is cancelled or executed, it is removed from the associated heap (while maintaining the heap property). Our event-loop is minimal and is designed to co-exist with the underlying JavaScript engine. We see our event-loop as an enhancement layer to add the essential adaptation mechanisms: priority and cancellation. Heaps in our design allow applications to queue events improving utilization while keeping full control over timing through prioritized execution and cancellation. To avoid blocking the underlying JavaScript engine, we can run DOHA's event-loop in a timed mode by setting a threshold (e.g., 200 ms) for the maximum duration of an event-loop iteration.

2.1.1 RAPT: Events and Policies

As a case study for DOHA, we choose the game Robots Are People Too (RAPT) [25]. RAPT won the most fun game award in Mozilla's Game On contest [21]. RAPT is an HTML5 platform game ported from C++. Players jump between moving platforms and coordinate their movements in order to pass game levels. The exit to each game level is blocked by enemies that roll, jump, fly, and shoot to prevent escape. RAPT uses 100% of a single core CPU (2GHz). To understand the time profile of different game components, we used the internal browser profiler. The performance of RAPT is impacted by two major components: graphics and simulation (physics and collision detection). In Chrome, 50% of the time is spent rendering, 30% on the simulation update, and around 20% is spent inside the browser. The major components (graphics, simulation, and AI) are similar to traditional desktop games [4]. We focused our experiments on the simulation updates because it constitutes a large performance concern especially after the rendering in browsers becomes hardware-accelerated.

```
function update (time) {
  var evt;
  // Delete pending events
  for(evt in pending_events){
    eventloop.cancel(evt);
  }
  // Add events to event-loop with a priority
  for(entity in game_entities){
    evt = new Event(entity.update, [time], BestEffort);
    evt.priority = entity.getPriority();
    eventloop.submit(evt);
  }
}
```

Figure 3: Modified simulation loop update call

Our first task was to split the large monolithic simulation loop into small explicit update events. The main simulation loop is now triggered by a timer event executing the global update function at a rate of 30 frames per second (33ms frame duration). Timer events triggering the global simulation update take precedence over best-effort events submitted within each frame. As shown in Figure 3, the modified global simulation update starts by cancelling the pending best-effort events from the previous frame. Then, a separate update event per game entity is created and submitted to the underlying event-loop. Before an update event is submitted, the getPriority policy method for each entity is called to calculate the event importance.

Adaptation policies developed with the game define relative importance among different game entities in each game loop iteration (game frame). Relative importance (priority) among game entities dictates the order of event execution. Since players are at the heart of a game, their updates are the most critical indicator of perceived game quality. Our basic adaptation policy assigns priority based on distance from active players. Priority is a number between 0.0 and 1.0. Players get priority 1.0. The priority assigned to the update events of other entities is inversely proportional to their distance from the closest player.

Using distance only can lead to starvation for distant entities. These entities will not be updated if resources are limited which causes flaws in their physics updates. To minimize starvation and ensure correct simulation for all game entities, we defined a minimum update heuristic based on the time since last update. As the time since last update increases, the priority increases to reach 1.0 when we exceed a maximum time between updates threshold. Finally, game entities have some non-linear behaviors, such as gravity. These behaviors limit scalability because they require a high and consistent update rate. Our policy needs to detect and account for these behaviors while assigning priority.

All our entities sub-class two base classes: enemy and player. These base classes define the adaptation policies other entities inherit. This current policy can be customized at run-time with the appropriate thresholds, such as minimum update threshold and distance ranges. We can also override a policy to include other factors specific to an entity type. For example, we can increase the priority of a bullet proportional to its speed.

It is important to note that with simple modifications to the main simulation update loop, it was possible to scale quality using DOHA's event-loop. Web-based games have other places where scalability can help trade accuracy for performance, such as the particle engine (visual effects accuracy) and AI logic (algorithm accuracy).

2.2 MultiProc: Concurrent Execution

HTML5 web workers are implemented using threads in major browsers and utilize multi-core hardware if available. Worker threads were envisioned to provide an API to run scripts in the background without locking the user interface [10]. Since their inception, web workers have been used in computationally expensive demo applications to speed-up highly parallel algorithms. For example, our parallel factorial micro-benchmark gets around 10x speed-up with 16 cores for large numbers ($3 * 10^9$ in Chrome).

We believe web workers have a larger role in enhancing the performance of interactive multimedia web applications especially in the mobile Web. Mobile platforms have low-end multi-core processors (e.g., 600 MHz) and browsing performance is the biggest barrier to entry for a large number of ambitious web applications. Using one of the most challenging web application domains, HTML5 games, we show that web workers with appropriate support can significantly improve performance and perceived quality.

2.2.1 MultiProc API

MultiProc provides mechanisms to write concurrent web applications with different architectures. As shown in Figure 4, we started with a central architecture that is tightly coupled. This architecture works in applications with minimal shared state. *remote_submit* is used to submit a remote event to the central scheduler. The scheduler (in the main thread) decides where to execute each event based on worker load statistics (orders workers based on load). Events can be cancelled using *remote_cancel* before they are assigned to a worker. To inform the main thread that an event was executed successfully, a worker calls *done*. This call updates the worker load statistics (number of active events).

Concurrent web applications with expensive communication (as shown in Table 2) are similar to distributed systems. To share state between application components across workers, MultiProc introduces a publish-subscribe communication API and RPC events. To send a direct RPC event to a specific worker bypassing the central scheduler, we use the *remote_direct_submit* call. Shared state can be published using *publish_state*. The state is transferred across worker boundaries and the method that subscribed for the state updates using the *subscribe_state* API call is notified.

2.2.2 Central Hierarchical Design

MultiProc started with support for a centralized master/slave web application architecture. The main browser thread is the master dispatching events to slave workers.

```
//Central scheduler API
remote_submit(Event e);
remote_cancel(Event e);
done(Event e);


//RPC and state management API
remote_direct_submit(Event e);
publish_state(topic, msg);
subscribe_state(topic, worker_id, function_name);
unsubscribe_state(topic, worker_id);
```

Figure 4: MultiProc public API

The main thread and workers each run their own event-loop to manage event execution. Worker creation, book-keeping, and scheduling decisions happen in the centralized scheduler. This central design is based on the observation that the main thread handles the Document Object Model (DOM) and that workers can only communicate with their parents (no direct communication between siblings). To extend our adaptation model across workers, events are queued in the main scheduler and sent according to their priority. DOHA's central scheduler allows a small fixed window of events inflight per worker. Upon notification that an event was executed successfully, another event is dispatched to the same worker. Since the window size is small, the scheduler is agile in responding to load imbalance among workers.

When an event reaches a worker, it is passed to the application code. The application code at the worker adds the event to the local event-loop which respects its priority. Each level of DOHA orders events according to their priority to approximate a distributed adaptive event-loop across workers. To balance load across workers, events are assigned to the worker with the smallest load (number of events in progress). MultiProc uses two heaps to manage remote events and workers. Remote events are ordered according to their priority while workers are ordered according to the number of active events.

Assuming each event can be executed in any worker, the centralized scheduler will have perfect load balancing. However, some events have dependencies (e.g., manipulate the same data-structure). To handle these dependencies between events our framework uses event *coloring* [27]. Programmers color events and MultiProc adheres to the coloring constraints. Events with the same color execute in the same worker and events with different colors can execute in parallel (in different workers). Workers can generate events and assign them unique colors. Since our design is centralized, a worker delegates the event to the main thread (master) which assigns it to the appropriate worker. Coloring is an easy to adopt [27] yet powerful concurrency control mechanism. If all events have the default color, we have a serial program. Having more colors reduces the scheduling constraints which leads to better load balancing across workers.

To test dynamic load balancing in the central design, we used a 3D animation that renders a large frame using ray tracing. Ray tracing is computationally intensive and has minimal shared state. To simulate a loaded worker, we limit the CPU share of one worker (out of 4 worker threads) to be 25% of the CPU time. To simulate dependency between events, we vary the percentage of events with unique colors. 100% means each event has a unique color (no dependency) and 0% means each event is colored with one of the four

Percentage Of Unique Colors	0	25	50	75	100
Delay (ms)	501	393	342	320	310

Table 1: Average delay to render a frame using ray tracing

major colors (25% of the total events per color to distribute work evenly across 4 workers). The original rendering application which uses round robin scheduling takes 309ms to render a frame when all the workers have enough resources and takes around 500ms when one worker is limited. In Table 1, we see that MultiProc central load balancing algorithm assigned events to other less loaded workers reducing the impact of the loaded worker and maintaining the same overall application execution time when each event has a unique color (100%). As the percentage of events with unique colors decreases (more dependencies between events), the rendering task gets delayed significantly. The load balancing logic was not able to run as many events in parallel and gets the same results as the original round robin version.

Our initial design assumed web applications have a central design where all execution events pass by the MultiProc scheduler. Even though our results with a simple application were positive, in the central design all application state accessed during each computation and the generated results cross worker boundaries. The communication cost become prohibitively expensive in complex applications, such as games, with tightly coupled components sharing state. Table 2 shows the high costs of a ping-pong message in HTML5 web workers as we vary the message size. 3ms is a relatively high cost considering the 33.3ms frame duration (or 16.6ms with a rate of 60 frames per second).

Message Size (bytes)	10	100	1K	10K	100K
Firefox (ms)	3	3	2.3	3	4.5
Chrome 15 (ms)	3.4	4	4.5	6	46.9

Table 2: Average delay for a ping-pong message between workers

2.2.3 State Management and Publish-Subscribe

To address these high communication costs, we moved to a less central design where the code in workers is more independent. We re-structured the simulation loop of RAPT as a network of components running in workers. As we see in Figure 5, each worker has an event-loop to run local events. Instead of sending all events and their related state across worker boundaries, we send few direct events (synchronization and control events) and necessary state updates between workers. For example, we send a game loop start iteration event from the main thread with minimum data parameters, such as the current time. Update events for entities assigned to the worker are generated locally and added to the local event-loop.

Without shared memory, workers can not access the browser Document Object Model (DOM). The code for each game entity in RAPT had to be split into two parts: one for simulation which runs in worker threads and another for rendering which runs in the main thread. The modified game loop performs rendering in the main thread while the loop in each worker performs the simulation. To maintain the game view, the rendering state of each entity is replicated. After each simulation update, each entity communicates the state needed for rendering back to the rendering replica in the main thread. This partial replication of the entity's state uses our publish-subscribe communication API, as we see in Figure 6. Partial replication transfers the minimum amount of state needed for rendering, such as the entity position (x, y), and orientation (angle).

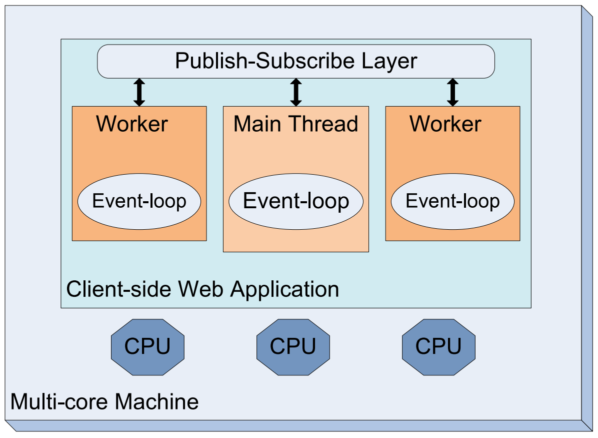

Figure 5: Web application using MultiProc with two workers

```
//Publish state in the worker
Entity.prototype.publishState = function()
{
   var msg = [this.x, this.y, this.angle];
   worker.publish_state(this.id, msg);
};
//Update state in the main thread
Entity.prototype.updateState = function(args)
{
   this.x = args[0]; this.y = args[1];
   this.angle = args[2];
};
```

Figure 6: Entity sharing rendering state in Concurrent RAPT using the Publish-Subscribe API

Few key entities in RAPT are global. For example, players are accessed and modified by different types of enemies in multiple workers. Similarly, some entities at the boundary of partitions need to have their state shared between two workers. To perform correct simulation, the entire entity's state is replicated across multiple workers. One worker owns the primary (authoritative) copy of the entity and other workers have full replicas. We synchronize all replicas after each entity update. The primary publishes its state to the entity's topic which all replicas (partial and full) subscribe to.

To allow modifying global objects, each identical replica acts as a proxy. State mutation is only allowed in the authoritative version of an entity. When a mutator method in a replica is called, the call is published on the global object mutation topic which the authoritative version subscribes to. State management in entities heavily use the publish-subscribe API for one-to-one (partial replication), one-to-many (full replication), and many-to-one (proxy forwarding) communication. These different communication patterns and the dynamic movements of entities to balance load across workers are the main motivations for our publish-subscribe communication API. Publish-subscribe provides a loosely coupled communication API that supports various communication patterns.

Our publish-subscribe logic is central. Web workers pass the communication API calls to the main browser thread. Our main publish-subscribe unit maps topics to a subscribers list. Each subscriber is a tuple of <worker ID, function name>. When a message is received from the topic, it is forwarded to a function with the given name on the specified worker. In each worker, the application registers a list of public functions that handle state-update messages.

The topics used for publish-subscribe communication need to be unique. We built a distributed identity manager to provide each game entity (in RAPT) with a unique identity that is used as a topic for its communication. The primary identity manager in the main browser thread assigns each worker a limited range in the identity space. When the identity range in a worker runs out, the remote identity manager asks the primary manager for a new range.

2.2.4 Load-Balancing

Our central scheduler implements load-balancing as we saw in Section 2.2.2. However, DOHA's state management support is agnostic to the way application components are distributed across workers. Building efficient distributed algorithms for games is an active area of research that is outside the scope of our work. We aim to provide the necessary mechanisms so application developers can implement their favorite distributed load-balancing algorithms on top of DOHA.

In the concurrent version of RAPT, we partition the game map geographically into a number of grids equal to the number of workers. Each worker handles a grid with all associated entities (enemies, and players). The state of each entity is updated in a single worker. This design respects data locality since each entity primarily interacts with other entities in its vicinity. Local interactions avoid expensive state transfer across worker boundaries. When entities move between grids, they migrate with all their state to a different worker.

Even though static geographical partitioning does not distribute work evenly across workers, our experience in few popular state of the art web-based games suggests that designers distribute game entities evenly across the game map. To help developers implement the load-balancing algorithms, DOHA provides:

- load information so developers can use it to decide when to migrate entities.
- a distributed identity manager which names entities uniquely, thus avoiding name conflicts upon migration.
- A loosely-coupled communication API to easily set-up and tear-down communication channels for frequent entity migration.

Developers need to develop the load-balancing policy and then use our communication layer to send the entity state.

DOHA aims to support applications with different concurrency requirements, ranging from simple applications that only need the computational benefits of web workers to the more demanding web-based games. Simple applications without shared state can delegate load balancing and scheduling of remote events to the central scheduler. For more advanced applications with shared state across workers, DOHA provides a publish-subscribe communication layer to manage state. In our efforts to parallelize RAPT, we initially tempted to isolate a major game component, such as the physics engine in a worker. This would have been easier and can probably enhance performance. However, it does not scale with the number of cores. Even though current mobile platforms have at most dual-core processors, RAPT and other web applications should aim for scalable parallelism to improve performance with more cores.

3. EVALUATION

We conducted a set of experiments with gaming scenarios of various computational demands. In the basic test map for RAPT, both players move inside a horizontal tunnel in one direction and the enemies move in a parallel tunnel above the players. We compare the following game versions: the original RAPT (RAPT), the modified RAPT using adaptation only (RAPT-A), and the modified concurrent RAPT (RAPT-C) with 2 web workers.

Our evaluation takes two views on performance: the first based on lower-level event-loop execution metrics, and the second based on higher-level application metrics. The low-level metrics include: number of events submitted per second, and the ratio of cancelled events. These low-level metrics show the throughput of the event-loop (events per second). To understand how these low level metrics affect game quality, we analyze the quality of the gameplay experience using high-level metrics, such as the simulation loop jitter profile (jitter median and jitter tail which is the 95th percentile of the jitter distribution) to quantify the average timeliness and the magnitude of execution glitches, and the average frames per second (FPS) versus priority for all entities to quantify the average game quality (scalable quality).

We performed our experiments on an AMD Opteron with 16 2GHz cores. Multi-core hardware allowed web workers to run on different cores. The duration of each experiment is 80 seconds. To avoid start-up and shutdown effects, we use the middle 60 seconds. We used Google Chrome 15.0.874.102 beta in Ubuntu 10.04 LTS. The two changing experiment parameters are the computational difficulty of the game scenario (which is controlled by the number and type of enemies) and the game version (RAPT, RAPT-A, and RAPT-C). We have three game scenarios: an easy scenario where all versions have reasonable quality; a medium scenario which is the hardest playable scenario by RAPT and RAPT-A (with the processing power of one core); and finally an extremely challenging game scenario with processing requirements beyond the capacity of one core.

3.1 Adaptive Execution

In this section we discuss the effects of our adaptation model on game performance. We analyze the low-level event-loop throughput, the timeliness of simulation loop updates, and the average overall game quality (scalable quality).

3.1.1 Timeliness

Table 3 shows the simulation loop jitter profile for all RAPT versions running all scenarios. The median jitter gives a measure of average timeliness and agility in responding to stimuli, such as input and collisions. To quantify glitches which affect quality negatively, we measure the jitter tail. The expected inter-arrival time between frames is 33.3ms since the target frame rate is 30FPS (frame duration 1000ms/30). We measure the offset for the expected arrival

Table 3: Jitter Profile

Jitter (ms)	Scenario	RAPT	RAPT-A	RAPT-C
Median	Easy	22	0	1
	Medium	47	0	0
	Hard	219	0	0
Tail	Easy	26	7	7
	Medium	55	8	17
	Hard	291	8	33

time and report its median and 95th percentile to capture the jitter distribution.

As seen in Table 3, the jitter median and tail in the original RAPT increases with the difficulty of the game scenario. The median jitter reaches 219ms which means RAPT executes 1 out of 7 frames (219/33.3=6.6) yielding a frame rate of around 4 FPS. This increasing jitter is due to RAPT's game loop iterating over all game entities in each frame leading to a large delay in processing each frame.

RAPT-A has low consistent jitter profile (median=0ms and tail=8ms) for all game scenarios. Our reactive event-driven design gives timer events more importance than best-effort events in each game loop frame. When the timer event running the game simulation loop (global update) fires, we stop execution of best-effort events and delete all pending events from the previous frame. Finally, RAPT-C has low consistent median jitter. But the jitter tail increases with the difficulty of the scenario to reach 33ms in the hard scenario (95% of the jitter values are less than 33ms). This increase in the jitter tail is mainly due to the communication and OS scheduling spikes for the two web workers.

3.1.2 Low-Level Event-loop Statistics

Table 4: Event Throughput Statistics

Statistics	Scenario	RAPT-A	RAPT-C
Event Submission Rate	Easy	7417	7127
	Medium	11150	10749
	Hard	33110	31317
Cancellation Ratio (%)	Easy	2.2	0.1
	Medium	17	18
	Hard	88	66

Our low-level event-loop statistics help us understand the event throughput. As seen in Table 4, both RAPT-A and RAPT-C submit more events as the difficulty increases because of the increase in the number of entities. The cancellation ratio also increases because the frame duration is not enough to update all entities as the difficulty increases.

RAPT-A submits slightly more events per second in all scenarios than RAPT-C indicating a higher simulation rate. The ratio of cancelled events in the easy and medium scenarios (RAPT-A and RAPT-C) is comparable. In the hard scenario, RAPT-C cancels less events (66%) than RAPT-A (88%). Moreover, RAPT-C has higher event rate due to having 2 extra cores to execute the simulation updates.

3.1.3 Priority Vs Quality

The medium scenario (hardest playable scenario for RAPT-A and RAPT) tests our capability to gracefully degrade quality when resources are scarce. RAPT in the medium scenario has an average quality of 12.3 FPS which is similar to the simulation rate. When resources are scarce, RAPT-A and RAPT-C cancel update events for stale low priority entities. Thus, the overall average game quality (in FPS) is not captured in the simulation rate (29 and 30 FPS).

To give more meaningful measure of the overall game quality (scalable quality), we measure the average frames per second for all game entities. We correlate this quality indicator with the priority assigned by our policy. As we see in Figure 7, the FPS of game enemies in RAPT-A ranges from 16 for low priority entities to 29 (maximum) for high priority entities. Similarly, the average jitter for all entities decreases from 17ms to 4ms as priority increases (average jitter is inversely related to average FPS). We notice that low priority entities never starve (have at least 16 FPS). This is due to our minimum update threshold which ensures that even low priority entities are updated at a lower frequency. When CPU is limited, our adaptation model in RAPT-A improves quality for important entities so quality has a strong correlation with priority.

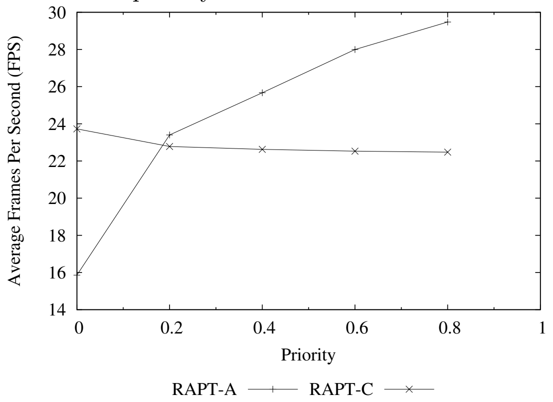

Figure 7: Priority Versus Quality (Average FPS)

RAPT-C has relatively higher quality for all entities (24 FPS). The quality does not have a strong correlation with priority because at each time instance game entities with the highest priority are concentrated in one worker (due to geographical partitioning of entities). Other workers at the same time instance are processing events with low priority leading average FPS to lose correlation with priority. In addition, the medium scenario with RAPT-C uses three separate cores (2 workers and the browser thread) and our adaptation scheme helps more when resources are limited.

3.1.4 Results Summary

RAPT-A and RAPT-C have better timing (lower jitter mean and tail) than the original RAPT. By cancelling updates of less important entities at the end of each simulation loop, RAPT-A and RAPT-C can provide important entities (with more influence over quality) a higher update rate. This translates to better overall game quality in RAPT-A and RAPT-C.

3.2 Concurrent Execution

While playing the game, we noticed RAPT-A had much worse perceived quality than RAPT-C in the hard scenario. RAPT-A's main thread was overwhelmed by the extremely high load and it was not yielding execution to the browser engine (to perform the rendering). In this case, isolation between the two tasks (simulation and rendering) in RAPT-C provided much better perceived quality.

The hard scenario is not playable in either RAPT or RAPT-A. RAPT-A was overwhelmed by the load and the simula-

Table 5: Jitter Profile in the Easy Scenario (with one core)

Jitter (ms)	RAPT	RAPT-A	RAPT-C
Median	25	5	17
Tail	28	8	28

tion rate in the original RAPT is extremely low (2 FPS). The hard scenario is only playable in RAPT-C. Even though the hard scenario had a large number of enemies, our design scales the communication costs. RAPT-C only executes and communicates state updates for as many events as the frame duration allows.

To evaluate the effects of adding more cores on game quality, we run RAPT-C using the hard scenario while varying the number of cores. As we see in Figure 8, the average FPS for all game entities (scalable quality) increases as we add more cores. RAPT-C with 1 worker gets an average of 3.5 FPS. With 3 workers, the average FPS is between 12 and 14. Average jitter also drops from 260ms with 1 worker to around 42ms with 3 workers. In the hard scenario, we get linear improvement in quality with each worker added up to 3 workers. As we see in Figure 8, using 4 workers does not improve the game quality. With 4 workers, only 3% of the execution events are cancelled which indicates the application load in the hard scenario is small relative to the available cores. In this case, RAPT-C pays additional concurrency costs but does not benefit from the extra core.

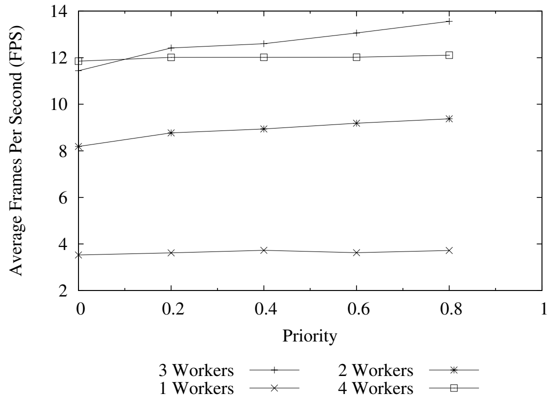

Figure 8: RAPT FPS in HARD scenario with 1, 2, 3 and 4 workers

To look at timeliness as we add more cores, we measure the jitter distribution of the simulation loop frames in the hard scenario. RAPT has by far the worst jitter profile. As we see in Figure 9, only 20% of execution frames have jitter less than 210ms. RAPT's jitter tail extends to around 291ms causing significantly bigger execution glitches. RAPT-A and RAPT-C (with 1 and 2 workers) have the same low average jitter profile and RAPT-C has a relatively worse jitter tail.

We also observe that latency increases as we add more cores. With 3 workers, the mean jitter is 14ms and the jitter tail is 64ms. The jitter increase is partly because the simulation loop in worker threads is triggered by a periodic update event sent from the main thread. When the number of workers increase, the communication load on the main thread increases and the loop update events are delayed.

3.2.1 Less Cores Than Workers

To test what will happen if we have more workers than cores, we ran the medium scenario in a single core machine

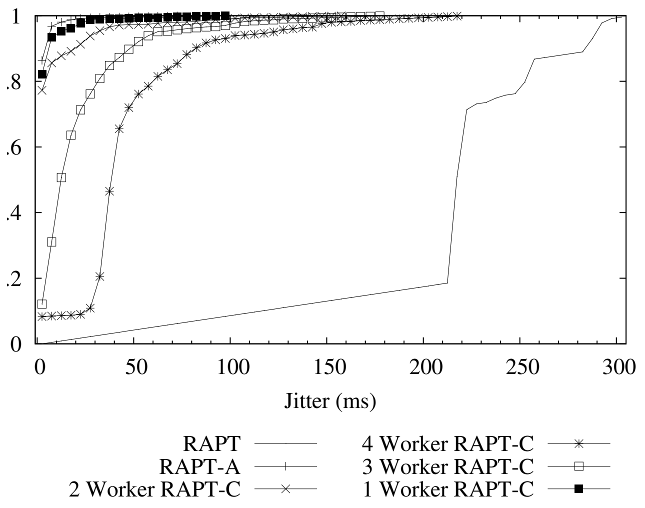

Figure 9: Jitter Cumulative Distribution in Hard Scenario

(a 2.80GHz Intel(R) Pentium 4). As we see in Table 5, RAPT-C has lower FPS and higher jitter tail than RAPT-A (but comparable quality to RAPT). We also noticed in the low-level event-loop statistics that RAPT-C submitted less events and cancelled more. This performance gap is due to the overhead of communication between and scheduling of the worker threads and the lack of any parallel speed-up using the one core machine.

Ideally we should have one worker per core. Degradation in performance is expected if we use more or less workers than necessary. To help applications choose the appropriate number of workers, browsers can have an API to expose the number of cores (user agent information) or JavaScript library developers can detect it (using micro tests).

3.2.2 Results Summary

DOHA demonstrates the potential to scale quality linearly as we use more cores in a challenging game scenario. It is essential to choose the appropriate number of workers for the application execution and communication load, and the underlying hardware. Using more workers than needed (4 workers case in Figure 9) or using more workers than cores (Table 5) can reduce performance due to concurrency overheads (without getting any parallel speed-up).

4. LESSONS LEARNED

This section includes few of the subjective lessons learned which can shed more light on DOHA and web-based game development. We noticed that:

- Performance engineering inside browsers is challenging. Browsers have primitive debugging and performance monitoring tools. Web workers have even less support. To conduct a rigorous experimental study and quantify performance, we had to build a complete performance analysis infrastructure. We built a parallel performance monitor to capture performance data from workers, and a visualization tool to display performance signals in real-time for interactive performance debugging. We are developing a tool to record and reply performance data and some execution state to enable interactive performance analysis offline.

- Game adaptation in RAPT, required minimal code changes. To introduce priority and cancellation of events we only had to change the core game loop as described in Section 2.1.1. Using the central scheduler was relatively easy having already used explicit events because most of the distribution tasks are delegated to the central scheduler. However, using web workers with a distributed application design requires re-structuring existing code in a major way. For example, RAPT was modified to have a distributed game loop and we used replication to manage shared state as described in section 2.2.3. Introducing the publish-subscribe layer improved the abstraction but developers still need to write complex distributed algorithms. Finally, parallel and adaptive execution are independent and can be adopted separately even though they are introduced together within the framework of scalable quality. Applications can use DOHA's publish-subscribe communication layer without adopting explicit events and the other way around.
- HTML5 web workers expose an elegant shared-nothing concurrency abstraction. Explicit message-passing is a good fit for asynchronous event-driven browser execution. It also allows web developers to use familiar distributed computing abstractions (from their experience with server components). The main limitation is the high communication costs in implementations. Current communication cost in browsers will enable applications to scale quality in multi-core platforms. To scale in many-core platforms, the communication costs needs to be reduced significantly. Browsers need to optimize the communication channels and expose optimization mechanisms (e.g., use immutable objects with ownership transfer to pass large objects across workers). JavaScript frameworks similar to DOHA can also perform communication optimization (batching and pipelining) to reduce messaging costs.
- DOHA is applicable in other web multimedia applications, such as video applications, visualizations, and animations. We observed the same event-driven architecture in the few animation and visualization platforms we studied. To extend our support to server-side game components, we ported DOHA to node.js [12], a popular JavaScript server framework. Our future work aims to use scalable quality in other application domains and perform an in-depth study of the adaptation and load-balancing policies required. Currently, our adaptation policies and the partitioning algorithms are developed separately. We think developing a load-balancing algorithm that is quality-aware will improve RAPT-C quality significantly. For example, it can distribute high priority entities evenly across cores to maximize their chance of getting updated.

5. RELATED WORK

DOHA builds upon the event-driven nature of popular web applications, which have a large number of short handler functions [22]. Event-driven programming is a natural fit for current JavaScript engines with single execution thread and asynchronous DOM APIs. Event-driven reactivity in DOHA has its roots in the concepts of reactive programming [3]. DOHA introduces event classes and specifies timing information at the event level similar to the application model in Cooperative Polling [14].

Unlike conventional multimedia adaptation techniques [16], our model does not require estimation of resource requirements, simplifying its usage drastically. DOHA application-level adaptation is inspired by priority-progress adaptation [15]. This adaptation technique was developed in multimedia video streaming with three main principles: incremental quality, priority assignment based on the contribution to perceived quality (temporal or spatial), and finally ordering computations according to priority. In DOHA, we developed CPU adaptation policies for web-based games and extended the adaptation model across parallel threads. To improve timing in networked applications, all resources (CPU, network, and storage) need to be considered [13]. Paceline [9] is a transport protocol above TCP that uses the same adaptation principles as DOHA to improve data communication timing in media-streaming applications. DonneyBrook's [6] interest sets use distance, aim, and recency from the player's perspective to decide which entities are more important. Similar to interest sets, our CPU adaptation policy uses distance from the player to determine the importance of game entities, but our priority scheme has a continuous spectrum (between 0.0 and 1.0) allowing smooth scalability instead of the two priority levels in DonneyBrook.

Recently, developers used web workers to separate the physics engine of a simple animation [17] improving the animation's frame rate. However, offloading functional units limits scalability to the number of independent units while DOHA aims for scalable parallelism using web workers.

Parallel game servers use techniques, such as Synchronization via scheduling [5] and Software transactional memory [18] to manage state. These techniques assume shared memory while web workers have no sharing and use message-passing. Similar to the Multikernel [2], we embrace the network nature of concurrent systems and re-structure our experimental web-based game as a network of distributed components. We use replication to share state using ideas from the distributed architecture of interactive multi-player games in Colyseus [7]. To manage concurrency in DOHA's central scheduler, we use coloring [27] which is a coarse grain technique that is easier to use than explicit dependencies between events in Grand Central Dispatch [11].

To improve the browser performance, the parallel browser project [19] re-writes the bottlenecks (parsing and rule matching) in a parallel fashion. Application-level concurrency is equally important especially with the slower pace of change in browsers. Native Client [26] allows web applications to execute native code inside a browser sandbox and improve performance with hand-coded assembler and native threads. DOHA aims to improve the performance of applications written in JavaScript, the de-facto language for web applications. Using native code in the browser is complementary to our work especially since JavaScript engines are becoming more mature. The exokernel browser architecture in Atlantis [20] defines a narrow API for basic services and allows web applications to extend their execution environments. Atlantis' run-time language, Syphon, supports a full threading model. Even though the performance of threads is arguably superior to web workers with message-passing, the performance gains come at the high cost of introducing a concurrency model that causes most systems errors [23].

6. CONCLUSIONS

Browsers are becoming mature platforms. Ambitious web applications with high computational demands and low latency interactions, such as games, animations, and interactive visualizations are pushing the limits of available processing resources. The best-effort execution model of current browsers lacks the necessary mechanisms to help applications control quality and balance between timeliness and utilization in overload conditions. Even though HTML5 web workers provide a concurrency model to utilize multicore resources, web developers still need more programming support to deal with hard concurrent software development issues, such as state management and load balancing.

With hints from application-level adaptation policies, DOHA favors important events that have more influence over application quality. To scale web application quality with widely available multi-core processors, DOHA simplifies concurrent real-time web development. When CPU resources are scarce, the modified game using DOHA had better timing and higher overall quality. More importantly, the quality scales linearly with a small number of cores. Scalable quality enables ambitious web applications to explore more challenging scenarios without the fear of brittle quality.

DOHA and the modified RAPT versions are open source and may be downloaded from `http://qstream.org`

7. REFERENCES

[1] Ambiera. Copperlicht - fast webgl javascript 3d engine. `http://www.ambiera.com/copperlicht/`. [accessed 30-Oct-2011].

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[3] G. Berry, G. Gonthier, A. B. G. Gonthier, and P. S. Laltte. The esterel synchronous programming language: Design, semantics, implementation, 1992.

[4] M. J. Best, A. Fedorova, R. Dickie, A. Tagliasacchi, A. Couture-Beil, C. Mustard, S. Mottishaw, A. Brown, Z. F. Huang, X. Xu, N. Ghazali, and A. Brownsword. Searching for concurrent design patterns in video games. Euro-Par '09, pages 912–923, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. *SIGPLAN Not.*, 46:640–652, June 2011.

[6] 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. *SIGCOMM Comput. Commun. Rev.*, 38(4):389–400, 2008.

[7] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. NSDI'06, pages 12–12. USENIX Association, 2006.

[8] M. Cook. Pistol slut. `http://pistolslut.com`, 2011. [accessed 3-May-2011].

[9] A. Erbad, M. Tayarani Najaran, and C. Krasic. Paceline: latency management through adaptive output. In *ACM MMSys '10*, pages 181–192, 2010.

[10] I. Hickson. Web workers. http://dev.w3.org/html5/workers/, 2009.

[11] A. Inc. Concurrency programming guide. `http://developer.apple.com/`, 2009. [accessed 9-Oct-2009].

[12] I. Joyent. Node.js: Evented i/o for v8 javascript. `http://nodejs.org/`. [accessed 2-Nov-2011].

[13] C. Krasic and J.-S. Légaré. Interactivity and scalability enhancements for quality-adaptive streaming. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 753–756, New York, NY, USA, 2008. ACM.

[14] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel. Fair and timely scheduling via cooperative polling. In *EuroSys '09*, pages 103–116, New York, NY, USA, 2009. ACM.

[15] C. Krasic, J. Walpole, and W. Feng. Quality-adaptive media streaming by priority drop. In *NOSSDAV '03*, pages 112–121, June 2003.

[16] R. Kuschnig, I. Kofler, and H. Hellwagner. An evaluation of tcp-based rate-control algorithms for adaptive internet streaming of h.264/svc. In *ACM MMSys '10*, pages 157–168, 2010.

[17] S. Ladd. Box2d, web workers, better performance. http://blog.sethladd.com/, 2011.

[18] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *EuroSys '10*, pages 41–54, New York, NY, USA, 2010. ACM.

[19] L. A. Meyerovich and R. Bodík. Fast and parallel webpage layout. In *WWW'10, Raleigh NC, USA*, 2010.

[20] J. Mickens and M. Dhawan. Atlantis: Robust, extensible execution environments for web applications. SOSP '11. ACM, 2011.

[21] Mozilla. High scores. `https://gaming.mozillalabs.com/games/winners`, 2011. [accessed 3-May-2011].

[22] B. Z. Paruj Ratanaworabhan, Ben Livshits. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *WebApps'10, Boston MA, USA*, 2010.

[23] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: taming device drivers. In *EuroSys '09*, pages 275–288. ACM, 2009.

[24] D. Szablewski. Biolab disaster. `http://playbiolab.com`, 2011. [accessed 3-May-2011].

[25] E. Wallace, J. Ardini, and K. Gishen. Robots are people too. `http://raptjs.com`, 2011. [accessed 3-May-2011].

[26] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53:91–99, Jan. 2010.

[27] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. MaziÃÍres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX ATC 2003*, pages 239–252. USENIX, 2003.