Soft Real-Time Scheduling in Google Earth

Jeremy P. Erickson *University of North Carolina at Chapel Hill*

Greg Coombe *Google, Inc.*

James H. Anderson *University of North Carolina at Chapel Hill*

Abstract

Google Earth is a virtual globe that allows users to explore satellite imagery, terrain, 3D buildings, and geo-spatial content. It is available on a wide variety of desktop and mobile platforms, including Windows, Mac OS X, Linux, iOS, and Android. To preserve the sense of fluid motion through a 3D environment, the application must render at 60Hz. In this paper, we discuss the scheduling constraints of this application as a soft real-time scheduling problem where missed deadlines disrupt this motion. We describe a new scheduling implementation that addresses these problems. The diversity of hardware and software platforms on which Google Earth runs makes offline execution time analysis infeasible, so we discuss ways to predict execution time using online measurement. We provide experimental results comparing different methods for predicting execution time. This new implementation is slated for inclusion in a future release of Google Earth.

1. Introduction

Google Earth is a 3D graphics application used by millions of consumers daily. Users navigate around a virtual globe and can zoom in to see satellite imagery, 3D buildings, and data such as photos and links to Wikipedia. An example screenshot is shown in Fig. 1. In addition to a free edition, professional editions exist for use by corporations and governments. Such editions are used in broadcasting, architecture, environmental monitoring, defense, and real estate for visualization and presentation of geo-spatial content. When navigating the globe, an illusion of smooth motion should be created. However, the application currently has a significant amount of visible *stutter*, a phenomenon in which the display fails to update as frequently as it should.

Several techniques have been proposed to address the problem of stutter in graphics applications. In [1] the technique of altering *level-of-detail (LOD)* is proposed to ensure consistent frame rates. The authors propose certain heuristics in order to estimate *rendering time*, the time required to render a single frame. In [12] these heuristics are explored in more detail and hardware extensions are proposed that can be used to enable soft real-time scheduling. Due to the hardware requirements, these techniques cannot be used for an application such as Google Earth that needs to run on a wide range of existing consumer devices. For the specific case of mobile devices, rendering time analysis methods are presented in [7, 10]. Although Google Earth does currently alter LOD, for this paper we focus on the complementary approach of delaying problematic jobs to future frames. Therefore, rather than predicting how the workload will be reduced when the LOD is reduced, we need accurate predictions for how long jobs will take to run. Furthermore, existing work applies detailed prediction mechanisms only to rendering jobs that execute primarily on the GPU, whereas we are also interested in CPU-bound and network-bound jobs.

Figure 1: Google Earth screenshot of St. Peter's Basilica from Sightseeing Tour.

In this paper, we provide methods to predict the execution time of these jobs. The wide variety of target platforms make offline analysis of execution time

impractical, so the execution time of jobs must be predicted based on online statistical collection. Furthermore, Google Earth has a limited preemption model (described in Sec. 2.1). These issues make reducing stutter difficult, particularly without negatively affecting the response time of jobs (see Sec. 2). The primary contribution of this paper is an implementation study of different methods to predict execution times for heterogeneous sets of jobs, with the goal of reducing stutter with minimum effect on response times. We propose a simple prediction mechanism that can significantly reduce stutter.

Our new scheduler implementation is planned for inclusion in a future release of Google Earth. Thus, it requires careful planning and testing to ensure that users, particularly paying customers, do not experience any loss of functionality. One difficulty in this regard is the requirement that the implementation be done "in-place" without disrupting the existing behavior (due to a planned production release halfway through this work). As changing the scheduling logic could have complex unforeseen consequences, the new scheduler must be capable of mimicking the existing behavior. This was complicated by the fact that the existing code was not designed with a clear notion of scheduling, resulting in several different ad-hoc methods.

In Sec. 2, we describe the terminology and task model used in this paper. Our description of the new scheduling implementation is given in Sec. 3, followed by descriptions of specific time-prediction algorithms in Sec. 4, and experimental results in Sec. 5.

2. Background

In Sec. 2.1, we describe the constraints that the application runs under and define relevant terms used in this paper. In Sec. 2.2 we provide an example schedule to motivate the task model, and in Sec. 2.3, we describe the task model.

2.1. Definitions

Each image rendered on the screen is called a *frame*. The illusion of smooth motion is created by rendering these frames at a sufficiently high rate. Due to the synchronization of commodity graphics hardware and the display device, the frame rate is a fixed amount (usually 60 Hz. for desktop machines). This means that all of the work to process and render a single frame must be completed in $1/60$th of a second, which gives a *frame period* of 16.67ms. A frame period boundary is known as a *vsync event* (from *vertical synchronization*). There is no benefit to completing rendering before the vsync event, but missing it introduces artifacts by redrawing the previous frame. This causes a noticeable discontinuity in the motion, known as *stutter*. See [8] for a discussion of the perceptual impact of stutter.

The Google Earth process consists of multiple *threads*, each with its own context and execution path, but all of which are part of the same process and share common memory. The first thread started by the operating system, and running the `main` function, is called the *main thread*. Due to the requirements of the graphics drivers on some systems that Google Earth supports, all direct access to the graphics hardware must take place within the main thread. For the purposes of this paper, we only consider the work executed on the main thread.

A *scheduler* is a unit of code that repeatedly calls certain functions based on scheduling constraints. Each function that is called constitutes a *task*, and each call to the function is a *job*. (A job can make its own function calls, which are considered part of the job.) A job is *released* when it is added to the scheduler for execution.

Each frame requires three stages of execution within the main thread, as depicted in Fig. 2. Initially, *static jobs* are run in a specific order dictated by data dependencies. These include traversing data structures to perform updates and culling, uploading data to the graphics card, handling input events, etc. An example of a typical frame is given in Sec. 2.2. Then, the *dynamic scheduler* considered here is invoked and runs *dynamic jobs*, described in more detail in Sec. 2.3. (Note: This work focuses on the dynamic scheduler. Therefore, when "scheduler" is used without qualification, it refers to the dynamic scheduler, and when "job" is used without qualification, it refers to a dynamic job.)

Finally, the *vsync function* is called. The purpose of the vsync function is to let the graphics hardware know that there is a new image ready for the vsync event, and it works by blocking until the next vsync event has completed and then returning. Observe that whenever the vsync function is started, it completes at the next vsync event, and starting the vsync function at the wrong time results in blocking the main thread for a large amount of time, as occurs in Frame 3 in Fig. 2. We say that a vsync event is *successful* if the main thread is running the vsync function when it occurs, and *unsuccessful* otherwise. To attempt to ensure that the next vsync event is successful, when the dynamic scheduler is run, it is given a *scheduler deadline* to complete by. The scheduler deadline is before the vsync event so that there is enough time for the overhead of returning from the dynamic scheduler and calling the vsync function. In the absence of overheads, the scheduler deadline would be at the vsync event.

From the user's perspective, a job's completion is

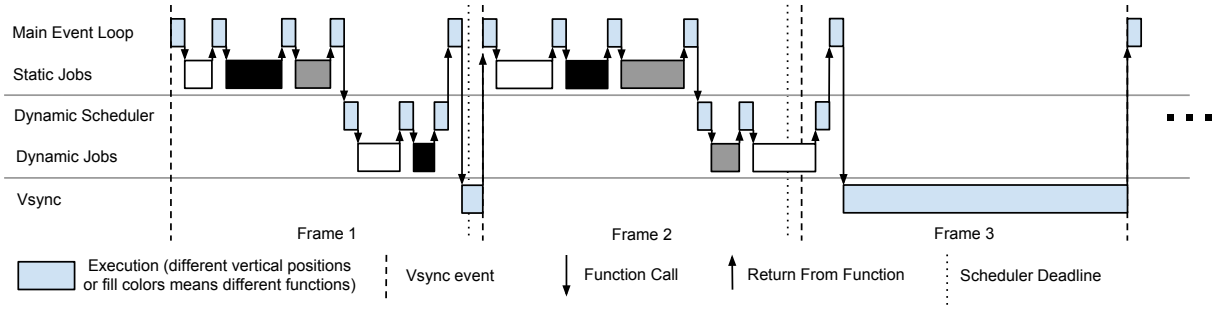


Figure 2: Execution path within the main thread. Sizes and numbers of jobs are simplified for illustration.

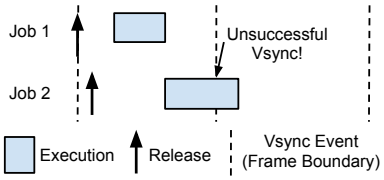


Figure 3: Both Job 1 and Job 2 have perceived response times of 2 frames, due to the unsuccessful vsync event caused by Job 2.

not visible until the next successful vsync event. Therefore, we define a job's *perceived response time* as the number of (complete or partial) frame periods between its release and the next successful vsync event after its completion, as shown in Fig 3. For example, if a job is released and completes between two vsync events, and the second vsync event is successful, its perceived response time is one frame period. The perceived response time of each job should be as short as possible. However, because stutter is more noticeable to the user than a delay of even a few seconds, avoiding an unsuccessful vsync event is generally a higher priority than achieving a small perceived response time. We discuss this tradeoff in more detail in Secs. 4 and 5.

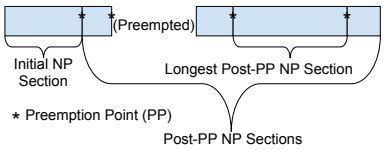


Figure 4: NP section types.

A point where a job can be preempted is called a *preemption point (PP)*, first described in [11]. (Our specific implementation of preemption points is discussed in Sec. 3.) We divide each job into *non-preemptive (NP) sections* based on its PPs. There are two *NP-section types*, depicted in Fig. 4. The first NP section of a job (from the start of the job until its first PP) is the *initial NP section*, and each NP section thereafter (between two PPs or between a PP and the job completion) is a *post-PP NP section*. Within the set of post-PP NP sections within a job, the longest (or an arbitrary longest in case of a tie) is referred to as the *longest post-PP NP section*. Each job contains exactly one initial NP section, zero or more post-PP NP sections, and at most one longest post-PP NP section.

A *predictor* is an object that has two functions: one that inputs task ID, NP section type, and NP section length for each initial or longest post-PP NP section and updates internal state, and one that outputs a prediction for the next NP runtime given task ID and NP section type. (The longest post-PP NP section is used to predict all post-PP NP sections. This decision is explained in Sec. 4.) A *predictor type* describes the particular method used to produce predictions given the inputs. Particular predictor types are described in Sec. 4.

A *soft deadline* is a deadline that can be missed, but should be missed by a reasonably small amount. There is still value in completing a job after its deadline has passed. Similar constraints have been discussed in [9]. A *firm deadline* is a deadline after which a job should not be completed, but should instead be discarded. Missing a firm deadline is costly, but not catastrophic.

2.2. Example Schedule

To understand the workload of Google Earth, it is helpful to understand some of the work needed to create a single frame for rendering. An example schedule (with parameters chosen for illustration rather than realism) is depicted in Fig. 5. Based on the camera position, the Visibility Computation creates a list of visible regions and compares it to the existing regions. If regions are not currently loaded or are at a different resolution, the same job queues up a fetch to request each needed region. In our example, there are two such

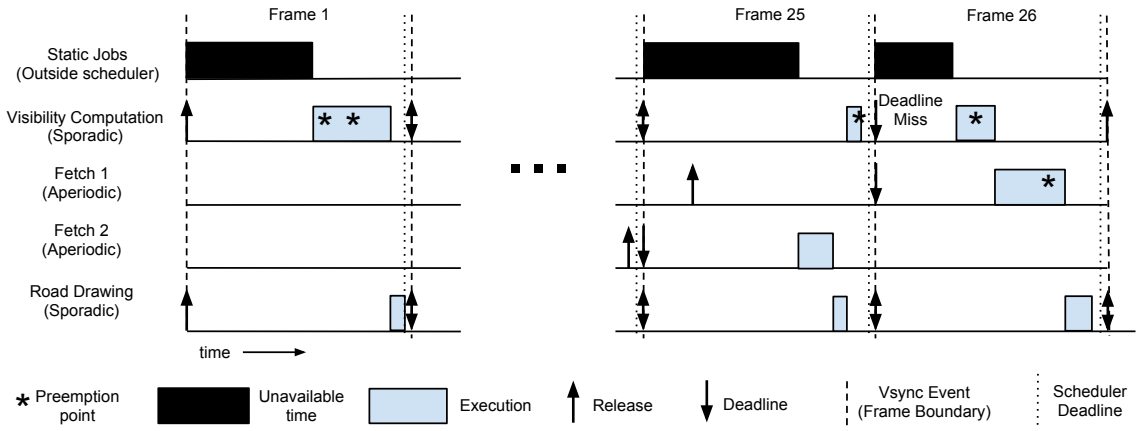


Figure 5: Example task system with simplified parameters chosen for illustration.

regions. When the server responds, each of these regions is processed (decoded, loaded into texture memory, inserted into the data structures, stored in a cache, etc.); in Fig. 5 this per-region processing is denoted as Fetch 1 and Fetch 2. This new data can cause a number of other data structures to be updated. For example, higher resolution terrain data requires updating the altitudes of the road segments — Road Drawing. Observe that, due to the preemption model, idleness is present at the end of each frame period to avoid missing the scheduler deadline (and, in turn, risking an unsuccessful vsync event). Also observe the large variance in execution times. For example, the Road Drawing task requires more work when new data has been processed. The amount of time left after the static jobs varies dramatically between frame periods. The dynamic scheduler must account for this variation.

2.3. Task Model

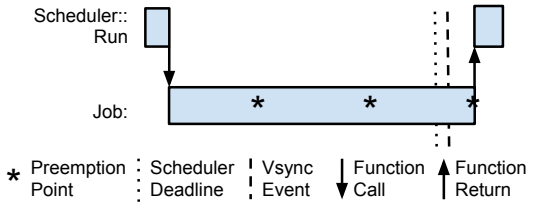
There are two types of tasks with dynamic jobs in Google Earth. A *single-active sporadic (SAS)* task has one job released at each vsync event, unless another job of the task has been released (at a previous vsync event) but has not yet completed. In our example discussed in Sec. 2.2 the Visibility Computation is an SAS task that misses its deadline at the end of frame 25, and so does not release a new job for frame 26. The Road Drawing task is also an SAS task, but meets all of its deadlines and therefore releases a job at every vsync event. The requirement that there be only one job that has been released but not completed is a simple form of adaptivity that limits the load of the system. (More complex adaptivity schemes exist, e.g. [4], but this simple scheme is used to ensure that each SAS task releases jobs as frequently as possible while having at most one job executing during each frame.) The soft deadline of each job is defined to be the next vsync event, so that in the ideal case each job has a perceived response time of one frame and a job is released at each vsync event. A *soft real-time (SRT) aperiodic* task, on the other hand, has jobs released in response to I/O events, such as receiving new imagery over the network. In the example discussed in Sec. 2.2, Fetch 1 and Fetch 2 are SRT aperiodic tasks. Because large amounts of data may be requested at the same time and network behavior is somewhat unpredictable, no constraints exist on the release times of such tasks. However, because new data usually indicates that the view on the screen needs updating, the timing of their jobs is also important. Therefore each job of an SRT aperiodic task also has a soft deadline at the next vsync event after it is released. A job of an SRT aperiodic task can be canceled if the application determines that it is no longer relevant. (For example, a network response with details about a 3D building that is no longer in view does not need to be processed.)

Furthermore, for analysis purposes, we can model the vsync event as a periodic task with an execution time of ε (for an arbitrarily small ε) and a period of 16.67 ms (assuming a 60 Hz refresh rate), with a firm deadline at the vsync event and released ε units before. This periodic task represents the time when the vsync function needs to be called. Any time that the vsync function runs past the vsync deadline is modeled as unavailable time like the static jobs that follow it. Meeting the deadline of the periodic vsync task corresponds to a successful vsync event, while missing it corresponds to an unsuccessful vsync event. If tasks were fully preemptible, meeting this task's firm deadline would be trivial, but due to the limited preemption model we must ensure

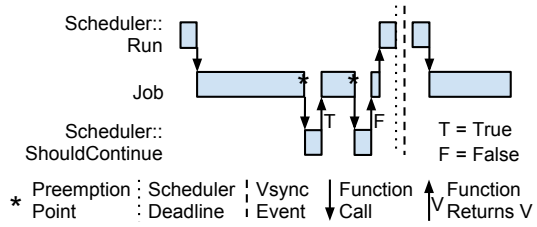
that no other job is running when the deadline occurs.

3. Implementation

In this section, we describe the implementation of a soft real-time scheduler within the legacy Google Earth codebase. First we describe, in Sec. 3.1, the existing implementation of task scheduling in Google Earth. Then, in Sec. 3.2, we discuss the new implementation.



(a) Existing scheduler relies upon jobs to do their own time accounting at their preemption points. This often causes unsuccessful vsync events.



(b) Jobs now contain explicit preemption points that call back to the scheduler through a function ShouldContinue. The scheduler then uses online prediction to determine whether to execute jobs, and if it returns "false," the job returns. If the job is unlikely to finish before the scheduler deadline, then it is postponed until the next frame period.

Figure 6: A similar job under the (a) old and (b) new schedulers. Moving the preemption control to the scheduler enables advanced prediction models.

3.1. Existing Implementation

In Google Earth version 6.0 and prior, the scheduler is passed the scheduler deadline, which is then passed to the jobs. Jobs are scheduled in simple FIFO order. The scheduler ceases executing when a job detects that it is past the deadline, as shown in Fig. 6(a). Jobs check for actual or expected deadline misses only at points where they can safely be preempted.

A small number of jobs use time prediction to avoid scheduler deadline misses. As these jobs process elements from a work queue, they maintain a running average of per-element execution times. If the average exceeds the available time before the deadline, they will voluntarily give up execution. This self-preemption ends the scheduler's execution as well, and no further jobs are scheduled before the vsync event.

3.2. New Implementation

For this work, we have implemented a new scheduling interface for Google Earth, which is included in a preliminary form in Google Earth 6.1 and will be included in a more complete form in a future release of Google Earth. We have a new interface `IJobScheduler`, as shown in Fig. 7. As with the old scheduler implementation, we continue to use FIFO to prioritize jobs. (Observe that, by our task model, FIFO is equivalent to EDF with appropriate tie-breaking.) Although the prioritization of jobs is identical in our new scheduler, the behavior has significant differences. For one, instead of relying on jobs to avoid scheduler deadline misses, the scheduler does not execute a job if it predicts it will cause a scheduler deadline miss, and stops executing jobs if the scheduler deadline is actually missed, as shown in Fig. 6(b). The new scheduler uses a predictor (predictors are defined in Sec. 2.1) to determine whether each NP interval is likely to cause a scheduler deadline miss. (If the current time plus the NP section prediction exceeds the scheduler deadline, then the job is predicted to cause a scheduler deadline miss.) Specific predictor types are discussed in Sec. 4. Unlike the previous scheduler, the new scheduler does not stop executing when the predictor indicates a scheduler deadline miss, but instead executes remaining jobs (with shorter predicted NP sections) that are not expected to cause scheduler deadline misses. In order to prevent starvation of jobs that have very long predicted NP section times, a non-starvation rule is applied: when the scheduler begins its execution for a particular frame, it always executes the highest-priority job (by FIFO), regardless of whether that job is predicted to cause a scheduler deadline miss.

A more direct preemption mechanism is also included in the `IJobScheduler` interface. When a job can safely be preempted, it can call a function `ShouldContinue` on the scheduler, optionally including a prediction for how much time it expects to run before it next calls `ShouldContinue` or completes. The scheduler will determine whether the job needs to be preempted and returns its decision to the job. This approach allows us to use existing PPs and/or time predictions where present, but we must modify all existing jobs by replacing deadline checks with calls to `ShouldContinue`.

```
class IJobScheduler {
  void AddJob(job);
  void RunJobs();
  bool ShouldContinue(job);

  class Job {
    bool Run();
    TaskID GetTaskID();
  };

  class TimePredictor {
    double Predict(
      task_id, np_section_type);
    void RecordExecutionTime(
      task_id, np_section_type, time);
  };
};
```

Figure 7: Interface for the implementation of new scheduler.

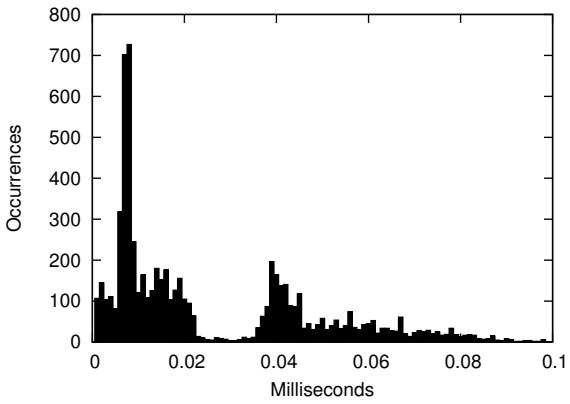


Figure 8: Histogram of initial NP section length for one task with the tail of the histogram truncated at 0.1 ms. The worst case for this task is 15 ms. Many of the tasks in Google Earth exhibit this long-tail behavior.

4. Time Predicton

As discussed in Sec. 3, the new Google Earth scheduler uses predictors to estimate NP section length. Choosing an appropriate predictor type is essential for performance. Here we see the competition between scheduler deadline misses and perceived response time: if the predictor is too optimistic, it can result in large numbers of scheduler deadline misses, but if it is too pessimistic, it can result in large perceived response times (because jobs are deferred to later frame periods). Scheduler deadline misses can also increase perceived response time by causing an unsuccessful vsync event, as jobs that completed before the unsuccessful vsync event will not have their effects observed until the next successful vsync event.

Accurate time prediction is complicated by the irregular distributions of NP section lengths for each particular task. A histogram of the initial NP section length for one particular task, measured from actual execution, is depicted in Fig. 8. The largest observed execution for this task is 15 ms, but its average execution is far smaller. If the predictor predicts shorter than 15 ms for initial NP sections of this job, then a scheduler deadline miss is possible. However, if the predictor predicts 15 ms or larger, then jobs of this task are likely to be delayed more frequently than necessary.

In this section we discuss several predictor types, with a brief description of the rationale for each. In each case, the scheduler also supports receiving time predictions from the job itself. If a job does provide such a time prediction (which is the case precisely for jobs that implemented time prediction in version 6.0), the scheduler will also call its associated predictor, and will operate based on the more pessimistic (i.e., larger) result. Pessimism can therefore only increase. We study the improvement available by adding predictions that did not previously exist, and are not concerned with the accuracy of predictions already present.

Because initial NP sections and post-PP NP sections are likely to involve different code paths, their timing statistics are handled separately. Code paths for different post-PP NP sections may or may not differ within a job, so we use the longest post-PP NP section when predicting any post-PP NP section time.

We now describe the predictor types tested in this work. Experimental results are given in Sec. 5.

No Scheduler Prediction. In order to mimic the behavior of the Google Earth 6.0 scheduler as closely as possible, one implementation of time prediction is simply to predict a time of 0 for each NP section of any type. If a job does not provide its own time prediction, then it will be scheduled unless the scheduler deadline has already been missed.

When using this predictor type, we sometimes modify the scheduler to return to the top level as soon as it decides not to schedule a particular job, to emulate the behavior of Google Earth 6.0 more precisely. (Recall, as discussed in Sec. 3, that the new scheduler will normally attempt to schedule a job with a shorter NP section prediction in this case.) This technique is referred to as *6.0 emulation*.

Maximum Observed Execution. For a worst-case execution time estimate, one metric is to use the maxi-

mum observed execution time for each type of NP section. While this method provides an obvious metric for "worst-case execution time," it has the problem that it is highly sensitive to outliers. For example, if the operating system decides to preempt Google Earth during the execution of a job, then the measured NP section length could be very large. Pessimistically assuming that all jobs of the same task can require such a large amount of execution time could result in large delays, and a large perceived response time for such jobs. Simple outlier detection methods are difficult to apply due to the irregular long-tailed distributions (as in Fig. 8) that these jobs exhibit even when functioning normally. Therefore, we instead allow values to expire, so as to limit the amount of time when a true outlier will have noticeable effects. For example, we can use the worst observed response time over the past six seconds instead of since the start of the application. If Google Earth is preempted while executing some job, then the relevant task will be penalized for only the next six seconds.

NP Section Time Histogram. If we minimize the probability of predicting too small an NP section time, then we risk creating large perceived response times. Instead, we consider the approach of targeting a specific small probability of such mispredictions. In order to do so, we can store a histogram of past NP section times. We use a bin size of 10^{-3} milliseconds, and allow a configurable percentile of the histogram to be selected. For example, if using a 95% histogram, the leftmost 95% of values will be selected, and the maximum value in the appropriate bin range used as an execution time estimator. The target in this case is a 5% probability of misprediction. Maintaining the histogram does require substantially more overhead than other approaches, but outliers are handled robustly.

Mean + Standard Deviations. Another predictor type uses the average response time of the job's NP section lengths of each type. We can calculate the mean and standard deviation of the previous values efficiently in an online manner. Using only the mean to predict NP section times would cause us to under-predict frequently (as roughly half of the previous times were longer), so we include several standard deviations above the mean. If task NP section distributions (for each NP section type) followed a standard distribution such as a normal distribution, we could compute the exact number of standard deviations necessary to achieve a specific percentile. However, because task NP section distributions are not consistent, doing so is not possible. By using the mean plus a certain (configurable) number of standard deviations, we attempt to predict with similar pessimism to the NP section time histogram, sacrificing precision for a significant reduction in overhead.

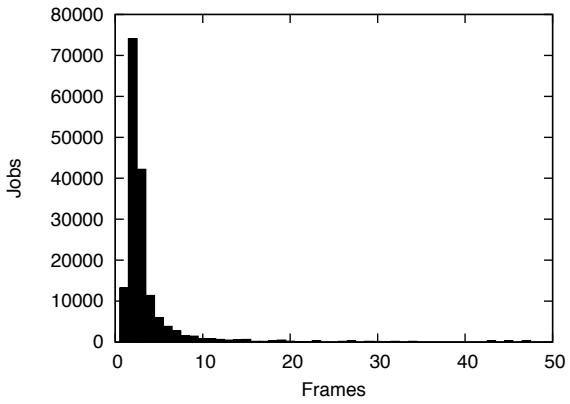


Figure 9: Typical perceived response histogram.

5. Experiments

In order to evaluate the tradeoffs between different predictor types, we performed experiments using a modified version of the "Sightseeing Tour" shipped with Google Earth. The Sightseeing Tour shows a variety of scenes across the planet such as the Eiffel Tower, St. Peter's Basilica (shown in Fig 1), downtown Sydney, and the Google campus in Mountain View, California. This particular tour makes heavy use of demanding features such as 3D buildings, so it exposes behavior that is likely to be worse than will be observed during typical interactive use. In the tour provided with Google Earth, there are pauses built into the tour at each major landmark, allowing the user to see the scene until they manually restart the tour. To facilitate testing, we removed these pauses from the tour, so that all landmarks are visited in one invocation.

For each run, we first cleared the disk cache of Google Earth, and then restarted the application. In this manner, we simulated the common situation in which the user visits a scene he or she has not previously visited. Furthermore, we ensured that each run began with the same cache state. This simulation therefore resulted in some of the most extreme overload conditions possible while running Google Earth.

We ran the tests on a Windows desktop (Windows 7 64-bit, Dell Precision T3400, 2.8GHz Intel Core2 Quad CPU, 8 GB RAM) and a MacBook Pro laptop (OS X Snow Leopard, a 2 GHz dual-core Intel Core i7 processor, and 8 GB RAM). Each test was based on the same version of Google Earth built from the current development repository, compiled with the same optimizations as production builds but including additional instrumentation for measurements. The viewport resolution was set to an HDTV value of 1280x720. Each experiment shown was repeated at least three times to ensure that

the results were representative of typical behavior. The performance differed on the two machines due to differences in OS behavior (such as scheduling), graphics cards, memory bus speeds, disk I/O speeds, and processor configuration (e.g. dual-core on the MacBook Pro vs. quad-core on the desktop).

Several factors can affect unsuccessful vsync events in Google Earth. It is possible for a frame to skip even if the scheduler completes before its deadline. There are two situations where this phenomenon can occur: the vsync event can be overrun by something that runs before the scheduler (a static job or the top level), or the scheduler deadline (which currently has an offset from the vsync event determined offline) can be too close to the vsync event. Similarly, it is possible for the scheduler to miss its deadline without causing an unsuccessful vsync event, if the scheduler deadline is farther than necessary from the vsync event. These considerations are outside the scope of these experiments, so we simply measure whether the last completion or preemption time occurs after the scheduler deadline.

When measuring perceived response time, we assumed that the next vsync event after the end of the scheduler would be successful, so that all tasks completed during that scheduler invocation had perceived completion times at the next vsync event. (This assumption follows from the definition of perceived response time given in Sec. 2.1. In the unlikely situation that the scheduler finished so close to the vsync event that there was not enough time to perform measurement, return to the top level, and call the vsync function, this assumption would be false.)

Because one or more job releases are cancelled for SAS tasks when a job misses its deadline, counting a job with a perceived response time greater than one frame as a single job biases our response metrics in favor of jobs with smaller perceived response times. For example, suppose within an SAS task we measure a series of 99 jobs that complete within one frame, followed by a single job that completes in 10,000 frames. Denote as I the time interval between the release and completion of the last job. In this case, 99% of jobs have a perceived response time of one frame. However, the actual performance is incredibly poor: I constitutes most of the time under consideration, but includes only one job completion for this task, and includes 9,998 cancelled releases. To avoid this bias, we account for each job J_s (for "skipped") that would have run, had its predecessor completed in time. We included such hypothetical jobs in our statistics, using for each J_s the perceived response time of the job J_r that was actually running when J_s would have been released. Most perceived response time histograms using this technique had the shape typified by Fig. 9, with variances in the length of tail.

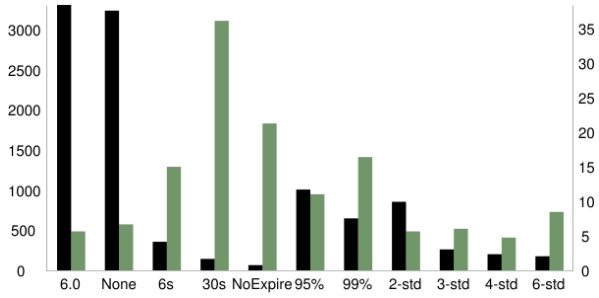


Figure 10: The tradeoff between Missed Scheduler Deadlines per Run (dark bars, axis on the left) and Average Response Times in Frames (light bars, axis on the right) for different time predictors, created from the Windows results in Table 1. Smaller is better.

Table 1 (with key features depicted in Fig. 10) shows typical results. By comparing the "No Scheduler Prediction" results with and without 6.0 emulation, we see that the minor modifications to the 6.1 scheduler, compared to the 6.0 scheduler, have very little effect on missed frames and response times. Adding scheduler-based time prediction, on the other hand, has a substantial impact on missed scheduler deadlines. This is illustrated in Fig. 10 by comparing the dark bars of the two left-most columns with the rest of the graph.

Selecting the maximum observed execution as the prediction for each type of NP section type provides the fewest scheduler deadline misses of any technique, as seen in the "Non-Expiring Maximum" results. However, doing so substantially increases response time for some tasks. Furthermore, although not depicted directly in the table, using a non-expiring maximum on Mac OS X results in over half of the aperiodic jobs being cancelled (because they involve work that is no longer relevant to the scene depicted by the time they would have been scheduled). Therefore, task starvation is significant, and the non-expiring maximum is not a practical choice for scheduling within Google Earth. (The difference in behavior between Mac OS X and Windows is due to the high sensitivity of this predictor type to outliers resulting from machine-specific behaviors.)

Expiring these values after 30 seconds significantly reduces the worst median response time, but maintains a large average perceived response time and worst perceived response time. It would appear that enough of the "worst" values expire quickly enough to avoid starving jobs, but the values expire slowly enough to cause delays. Using 6-Second Maximum further improves the response statistics at the expense of more scheduler deadline misses. This phenomenon occurs because a

Predictor Type	Platform	Missed Scheduler Deadlines Per Run	Worst Median (Within Task) Response (Frames)	Average Response (Frames)	Worst Response (Frames)
No Scheduler Prediction with 6.0 Emulation	Windows	3260	3	4.4	107
	OSX	3164	2	6.5	220
No Scheduler Prediction without 6.0 Emulation	Windows	3190	3	5.2	238
	OSX	2919	2	4.0	107
6-Second Maximum	Windows	351	3	11.7	372
	OSX	1072	2	7.6	193
30-Second Maximum	Windows	141	3	28.2	741
	OSX	614	3	9.8	760
Non-Expiring Maximum	Windows	64	30	16.6	619
	OSX	8	8126	270.5	16006
95% Histogram	Windows	992	3	8.6	237
	OSX	1062	2	14.7	357
99% Histogram	Windows	638	3	12.8	439
	OSX	417	3	7.3	209
Mean + 2 Standard Deviations	Windows	842	3	4.4	112
	OSX	887	2	3.5	67
Mean + 3 Standard Deviations	Windows	257	3	4.7	103
	OSX	450	2	4.1	110
Mean + 4 Standard Deviations	Windows	198	3	3.7	107
	OSX	581	2	3.8	98
Mean + 6 Standard Deviations	Windows	173	3	6.6	317
	OSX	552	3	3.8	283

Table 1: Typical run of a tour for each predictor type/platform combination.

small fraction of jobs take a disproportionate amount of time. These long-running jobs cause unnecessarily large predictions, resulting in other jobs within the task being unnecessarily delayed. When predictions are allowed to expire, response times improve and starvation is reduced.

A finer-grained level of control is available by maintaining an explicit histogram of non-preemptive execution times, and predicting based on percentile. Using a 95% Histogram significantly reduces missed scheduler deadlines, but not to the same degree as other methods we attempted, and response times were actually higher than with several other methods. A 99% Histogram provides more significant reduction in missed scheduler deadlines, but like the 95% Histogram has a mediocre response time distribution. This is likely due to the overhead of maintaining the histogram and computing the percentiles. Measurements on the OSX laptop showed that the histogram-based techniques required $2\times$ to $10\times$ more time to compute predictions than other techniques ($\approx$ 0.4ms to 2ms per frame).

The most promising method is to compute the mean and standard deviation for each distribution, and add a specific number of standard deviations on top of each mean. This is illustrated in Fig. 10 with "Mean + 2 Standard Deviations" through "Mean + 6 Standard Deviations". Because the different tasks do not follow a consistent distribution, as discussed in Sec. 4, we relied on experimental results to determine an appropriate number of standard deviations. Selecting values three or four standard deviations above the mean appears to be the best choice we tried on both tested platforms, providing both a small number of scheduler deadline misses and small response times.

This result was surprising and a bit counter-intuitive. For one, this prediction type does not expire any values, so it would seem to be susceptible to the same problems as the Non-Expiring Maximum, albeit to a lesser degree. In addition, the variability of response times would seem to favor the more complex histogram methods over "Mean + Standard Deviation". Our experiments show that this is not the case. Not only

does it outperform all of the other predictor types, but it is fast and requires very little storage. This makes it a good choice for use in a future version of Google Earth.

6. Conclusion

In this paper we have described a new implementation of a soft real-time scheduler in Google Earth. We have provided methods to dramatically reduce the number of missed scheduler deadlines, and demonstrated these results experimentally. In addition, these methods provide valuable insight into the execution behavior of jobs in Google Earth. This will greatly assist the developers in reducing stutter, particularly as more of the static jobs are made preemptible, decoupled from data structures, converted to dynamic jobs, and placed under control of the dynamic scheduler.

In this paper we focused on single-core scheduling, but with the proliferation of multi-core devices, multi-core scheduling is the next challenge. In that setting, the scheduler will manage a pool of threads and assign jobs to threads. The size of the pool will be based on the number of cores, device capabilities, and type of job (I/O blocking, requires access to graphics hardware, etc). Initially, we will focus on static assignment of jobs onto the main thread vs. non-main threads, but we have considered dynamically assigning jobs to threads based on workload by using a global scheduling algorithm. Our existing FIFO scheduling algorithm is appealing when extended to multiprocessors, because it is *window-constrained* as described in [3]. As that work demonstrates, bounded tardiness is thus guaranteed provided that the system is not over-utilized. (Similar work, e.g. [5, 6], demonstrates that similar results are possible when the system is not over-utilized *on average*, as we expect to be the case for Google Earth.)

The statistical predictors can also be improved. Predictors such as Mean can be very sensitive to outliers, particularly for a small number of samples. The Mean + Standard Deviation predictors do not currently expire any values, which was beneficial for the Maximum Observed Execution predictors. We would also like to investigate better statistical predictors for long-tail distributions, as mean and standard deviation are intended for normal distributions. The Histogram predictors are robust, but the cost of maintaining the data structures and computing percentiles (particularly for the sparse long-tail) is too expensive. This could be addressed through algorithmic improvements or by using approximate histogram methods [2]. We would also like to quantify the mis-predictions (when we assumed a shorter or longer execution time than actually occurred) directly and use this data to improve the predictors.

References

[1] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 247–254, New York, NY, USA, 1993. ACM.

[2] A.C Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M.J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Annual ACM Symposium On Theory Of Computing*, volume 34, pages 389 – 398. ACM, 2002.

[3] Hennadiy Leontyev and James H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44:26–71, March 2010.

[4] Chenyang Lu, John A. Stankovic, Tarek F. Abdelzaher, Gang Tao, Sang H. Son, and Michael Marley. Performance specifications and metrics for adaptive real-time systems. In *RTSS*, pages 13–23, Washington, DC, USA, 2000. IEEE Computer Society.

[5] Alex F. Mills and James H. Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *RTAS*, pages 311–320, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Alex F. Mills and James H. Anderson. A multiprocessor server-based scheduler for soft real-time tasks with stochastic execution demand. In *RTCSA*, pages 207–217, Washington, DC, USA, 2011. IEEE Computer Society.

[7] Bren C. Mochocki, Kanishka Lahiri, Srihari Cadambi, and X. Sharon Hu. Signature-based workload estimation for mobile 3d graphics. In *DAC*, DAC '06, pages 592–597, New York, NY, USA, 2006. ACM.

[8] Ricardo R Pastrana-Vidal and Jean-Charles Gicquel. A no-reference video quality metric based on a human assessment model. In *Proc of the Fourth International Workshop on Video Processing and Quality Metrics for Consumer Electronics*, 2007.

[9] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004. 10.1023/B:TIME.0000045315.61234.1e.

[10] Nicolaas Tack, Francisco Morán, Gauthier Lafruit, and Rudy Lauwereins. 3d graphics rendering time modeling and control for mobile terminals. In *Web3D*, Web3D '04, pages 109–117, New York, NY, USA, 2004. ACM.

[11] Jack P. C. Verhoosel, Dieter K. Hammer, Erik Y. Luit, Lonnie R. Welch, and Alexander D. Stoyenko. A model for scheduling of object-based, distributed real-time systems. *Real-Time Systems*, 8:5–34, 1995. 10.1007/BF01893144.

[12] Michael Wimmer and Peter Wonka. Rendering time estimation for real-time rendering. In *EGRW*, EGRW '03, pages 118–129, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.