A Soft Real-Time, Parallel GUI Service in Tessellation Many-Core OS *

Albert Kim, Juan A. Colmenares, Hilfi Alkaff and John Kubiatowicz
Parallel Computing Lab, CS Division, UC Berkeley
{alkim,juancol,hilfia,kubitron}@eecs.berkeley.edu

Abstract

We discuss the design and implementation of a parallel GUI Service in Tessellation OS and investigate its capability to provide soft service-time guarantees to visual applications. Use of CPU bandwidth reservation permits our GUI service to miss only 0.1% of client deadlines under an overloaded scenario, while a more traditional windowing system misses over 50% of its deadlines. Further, the GUI Service's parallel architecture nicely exploits parallelism to reduce service times for client requests.

1 Introduction

The trend of increasing the number of cores in client devices continues unabated. Unfortunately, the user's experience is not always improved by additional cores. In particular, the behavior of graphical user interfaces (GUIs) has remained relatively unchanged. Users are often frustrated because they expect visual interfaces with reduced and consistent response times to user actions; instead they encounter jumpy, sluggish behavior. They expect responsiveness even while running a simultaneous mix of interactive, real-time, and high-throughput parallel applications; instead, they confront waiting icons such as hourglasses and beach balls. Although such application mixes are becoming the norm, they are not well supported by today's commodity operating systems (OSs).

A key element to meeting user expectations is a GUI subsystem capable of providing differentiated service and service-time guarantees as well as high performance through parallelism. In this paper, we present such a GUI subsystem. It is implemented as the *GUI Service* of Tessellation OS [5], which is an experimental multi-core OS focused on enforcing resource allocation guarantees for client applications. An overview of Tessellation is given in Section 2.

*Research supported by Microsoft Award #024263, Intel Award #024894, matching U.C. Discovery funding (Award #DIG07-102270), and DOE ASCR FastOS Grant #DE-FG02-08ER25849. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of their sponsors.

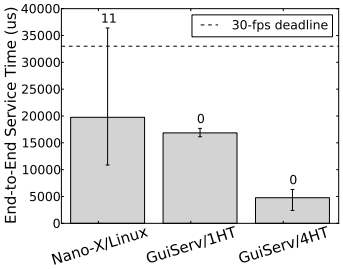

Figure 1: Service times of Nano-X on Linux and Tessellation's GUI Service using 1 and 4 hardware threads (HT) for expensive video requests. Above each bar is the number of missed deadlines.

Tessellation combines *space-time partitioning* and *two-level scheduling* to provide performance predictability in an efficient, scalable manner. In Section 3, we discuss how our GUI Service takes advantage of Tessellation's high degree of performance isolation and customizable user-level schedulers to provide service guarantees. A customized *Constant-Bandwidth Server* [3] forms the core of the request handling logic. We believe that the GUI Service is an embodiment of a general software architecture for QoS-aware, parallel services in Tessellation, and we are using it as a model to build the Network and File Services.

In addition to providing guaranteed behavior to existing applications, the GUI Service must perform admission control. Traditional admission-control tests require knowledge of clients' timing behaviors, which are often very difficult to determine. Instead, we propose that the GUI Service perform online profiling of visual applications to estimate their timing models. Based on these models the service can suggest Service Level Agreements (SLAs) to the applications.

In Section 5, we compare the performance of Tessellation's GUI Service and the Nano-X Window System[1], the latter representing the traditional, single-threaded approach to window systems. We show that the GUI Service is able to provide more predictable service times than Nano-X and also offers reduced service times by exploiting task parallelism. Figure 1 provides a sneak-peak at our results.

[1] http://www.microwindows.org/

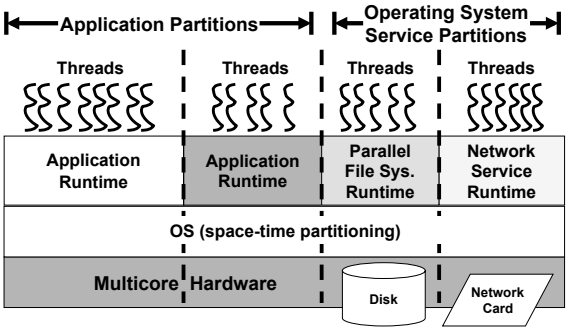

Figure 2: Space-time partitioning (STP) in Tessellation OS: a snapshot in time with four spatial partitions.

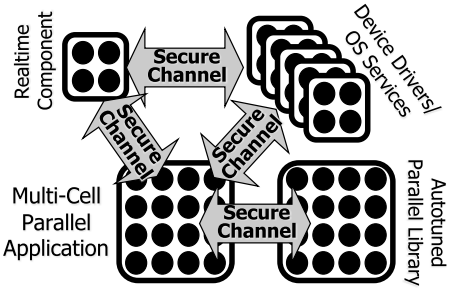

Figure 3: Decomposing an application into a set of communicating components and services running with QoS guarantees within cells. Tessellation OS provides cells that host device drivers and OS services.

2 Overview of Tessellation OS

Tessellation [5] is built on two complementary design principles [17, 12]: *space-time partitioning* and *two-level scheduling*. Space-time partitioning (STP) provides performance isolation and partitioning of resources among software components. Tessellation divides the hardware into a set of simultaneously-resident (spatial) partitions as shown in Figure 2. Partitions are virtualized and exported to applications and OS services through an abstraction called a *cell*, which is a user-level container that gives guaranteed access to resources for a set of parallel software components. An application running within a cell behaves as it would when executing on a dedicated machine. Two-level scheduling, on the other hand, separates global decisions about resource allocation to cells (*first level*) from application-specific management and scheduling of resources within cells (*second level*).

The Cell Model. Applications in Tessellation are divided into performance-isolated cells that communicate through efficient and secure *channels* (see Figure 3). Once resources have been assigned to cells, the user-level runtime within each cell may utilize the resources (e.g., hardware-thread contexts and memory pages) as they wish – without interference from other cells. The cell's second-level runtime can thus be customized for specific applications or application domain, for instance, with a particular scheduling algorithm and page replacement policy. Furthermore, the model allows cells to contain one or more address spaces (protection domains).

Inter-cell communication is restricted to channels. Channels (once constructed) enable fast asynchronous message-passing communication at user-level. The setup and tear-down of a channel is privileged and strictly controlled by the OS.

Enabling Performance Isolation. Each cell, when active, is allocated a partition, which is a performance-isolated unit of resources. Partitionable resources include hardware-thread contexts, pages in memory, and guaranteed fractional services from other cells. They may also include guaranteed cache slices, portions of memory bandwidth, and fractions of the energy budget, when the required hardware mechanisms are available (e.g., [16, 10]).

Cells may be time-multiplexed, as implied by the "time" component of the term space-time partitioning. Hardware-thread contexts and other resources are, however, *gang-scheduled* [14] such that cells are unaware of this multiplexing; i.e., unexpected virtualization of physical resources does not occur.

Tessellation provides several time-multiplexing policies for cells, some of them offering high degrees of time predictability; they are: 1) *no-multiplexing policy* (cell given dedicated access to cores), 2) *time-triggering policy* (cell active during predetermined time windows), 3) *event-triggering policy* (cell activated upon event arrivals, but never exceeds its assigned fraction of processor time), and 4) *best-effort policy* (cell with no time guarantees). Tessellation incorporates admission control to prevent cells from compromising the timing behavior of other cells.

Support for STP consists of a combination of hardware and software mechanisms. In this regard, Tessellation kernel has some similarities to a hypervisor (e.g., [2]), but with a crucial difference: its sole task is to provide performance-isolated, QoS-guaranteed containers for applications and OS services.

Adaptive Resource Allocation. Since applications' demands may vary over time, the resources assigned to cells may vary as well. Tessellation attempts to strike a balance between maximizing resource utilization to achieve performance goals and selectively idling or deactivating resources to provide QoS guarantees. Resource redistribution occurs at a coarse time scale to amortize the cost of the decision-making logic and to allow time for second-level scheduling decisions to be effective. The decision-making logic is packaged into a *Policy Service* that distributes resources to cells; details appear elsewhere [5].

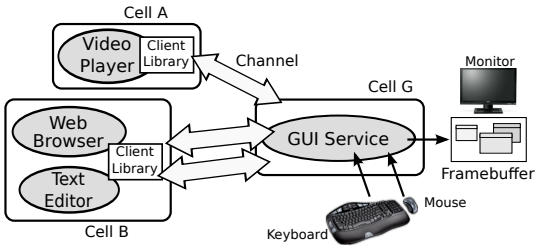

Figure 4: Tessellation's GUI Service in its dedicated cell and interacting with applications via channels.

OS Services. Cells provide a convenient abstraction for building OS services such as device drivers, network interfaces, file systems, and window systems. OS services can reside in dedicated cells, have exclusive control over devices, and encapsulate user-level device drivers. Tessellation adopts a philosophy similar to that of microkernels [11]. Unlike traditional microkernels, however, multiple cells can be mapped to the hardware simultaneously – allowing rapid inter-domain communication. Further, each cell is explicitly parallel and performance-isolated from the others.

Partitioning OS functionality into a set of interacting cells helps provide predictable and reliable behavior due to limited interaction with the rest of the system. QoS guarantees on shared services can be enforced by restricting channel communication. Alternatively, the capacity of overloaded services can be increased by resizing cells.

3 GUI Service

Tessellation's GUI Service (*GuiServ*) offers general window management, with graphical, video, and image processing services to applications. It exploits task parallelism for improved service times, while at the same time providing differentiated service and soft service-time guarantees to graphical applications.

GuiServ exemplifies Tessellation's view of QoS-aware, parallel services. As shown in Figure 4, *GuiServ* resides in a dedicated cell with sole control of the video output devices (e.g., framebuffer) and human-interface devices (e.g., keyboard and mouse). Visual applications communicate with *GuiServ* through inter-cell channels. Both client requests and input event notifications travel through these channels. A client library facilitates development of visual applications; it offers a friendly, high-level API to manage connections and interact with *GuiServ*.

GuiServ provides reserved fractions of processor capacity and performance isolation to visual applications. Applications demanding service-time guarantees *agree on* and *respect* certain conditions for using *GuiServ*. Typical conditions are, for example, a maximum inter-arrival rate (MIR) for service requests, specified as the maximum number of requests in a time period, and maximum number of outstanding requests at any given time. The desired service-time bound, capacity reservation, and conditions of use can all be included in the Service Level Agreement (SLA) between an application and *GuiServ*.

In Section 3.1 we propose a simple, yet practical approach to establishing SLAs between visual applications and *GuiServ*. This process is one of discovering the underlying system resources required to guarantee the behavior desired by clients. We follow in Section 3.2 by describing how we utilize the features of Tessellation OS to enforce these SLAs and provide soft real-time guarantees to graphical applications. As an additional benefit, SLA enforcement helps to make *GuiServ* immune to denial-of-service (DoS) attacks.

3.1 Providing SLAs for the GUI Service

Traditional real-time systems utilize static performance models of their software components to facilitate proper scheduling of client threads. Developers must test, analyze, and profile their applications to calculate their application's demands from the system. This traditional approach is complex, time-consuming, and brittle (under changes in system resources). Since we are targeting a more general pool of developers and target systems, it is not practical to assume the presence of static, a-priori performance models.

We propose to use an alternative approach: an online profiling API for establishing resource requirements. In this approach, clients start with non-real-time, best-effort guarantees from *GuiServ*.[2] Once a client has initiated contact with *GuiServ*, it may start marking some of its requests as *probe requests*, namely those for which the client wishes to achieve real-time guarantees. *GuiServ* runs these requests on some reserved CPU bandwidth (e.g., 10%) to estimate their resource demands.

After sending enough probe requests, the client may ask *GuiServ* about the guarantees that it can provide to the client. *GuiServ* will calculate the amount of CPU bandwidth it has left, and using the data gathered from profiling the probe requests, it will determine the service time that it can guarantee for each type of probe request, and report the results to the client. If the client finds the proposed terms acceptable, these guarantees will be formalized as an SLA. *GuiServ* may refuse to report any service times or CPU bandwidth if it deems that it has not procured enough data from the probe requests or if it does not have enough resources to make guarantees.

Although this approach may be ill-suited for hard real-time tasks, it provides a practical and relatively

[2] *GuiServ* reserves some CPU bandwidth (e.g., 40%) for best-effort clients, processing their requests in round-robin fashion, much as a traditional windowing server.

easy way for the common developer to request soft real-time guarantees for its applications without having to manually profile them.

3.2 GUI Service's Software Architecture

Figure 5 shows the basic architecture of *GuiServ* that allows it to guarantee SLAs with clients once they have been established. In the following, we discuss the specifics, including *Rendering Tasks*, responsible for rendering graphics and video processing requests, *Input Event Handler Tasks*, which forward input events from human-interface devices to clients and/or the service, and the *Window-Manager (WM) Task*, responsible for window creation, movement, and destruction. Other details are omitted due to space limitations.

Rendering Tasks. Rendering tasks do the heavy lifting in *GuiServ* and can be computationally expensive. They also have a primary role in providing guarantees to clients. To enable parallel execution, *GuiServ* dedicates a channel, a Rendering Task and a *virtual framebuffer* (VFB) to each client application. Since Rendering Tasks serve different clients, they can execute in parallel without need to coordinate among themselves.

Clients group requests into atomic "actions" which represent consistent updates to a client window (e.g., a complete video frame). SLAs reference actions, rather than individual requests. Each Rendering Task processes requests from input channels in FIFO order and updates its VFB. Once the last request for an action is processed the task clips the graphics content of its VFB (based on the location of its window on the screen), then copies the resulting content to the real framebuffer.

To provide guarantees to clients, Rendering Tasks are scheduled as *Multiprocessor Aperiodic Servers* (MASs) [4] under the Multiprocessor Constant-Bandwidth Server (M-CBS) scheme [3]. The MAS/M-CBS reservation scheme ensures each Rendering Task RT_i a *fraction of processor capacity* (specified by two parameters: a worst-case utilization U_i and a period P_i) and *isolation* from the effects of other tasks. Using capacity reservations for Rendering Tasks prevents any application from monopolizing *GuiServ*.

A Rendering Task, if not active, can be triggered upon the arrival of request messages in the input channel. The mechanism by which this triggering occurs is a vital aspect of providing guarantees. For instance, if Rendering Tasks are awoken directly by message arrival, the channel notification mechanism must limit the number and frequency of requests that clients can inject, so that no client exceeds the parameters in their SLAs. An alternative is to have a *Front-End Task* that periodically inspects the channels and spawns Rendering Tasks, if required, while restricting the processing of requests from clients that abuse their SLA limits. The Front-End Task can be scheduled using global Earliest-Deadline-First (EDF), which is compatible with the use of MAS for Rendering Tasks.

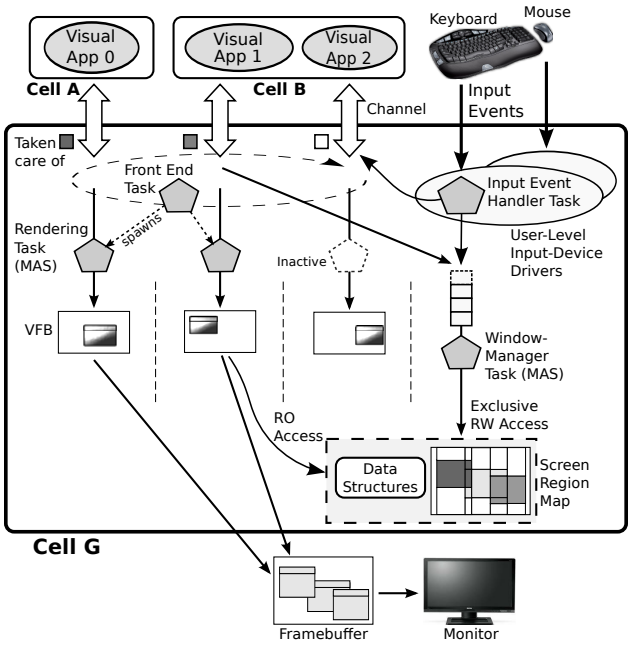

Figure 5: Software architecture of the GUI Service. Parallel *Rendering Tasks* process client requests while honoring client SLAs. Results placed into *virtual framebuffers* are later clipped and combined to produce final image.

Input Event Handler Tasks. The Input Event Handler Tasks are part of the user-level input-device drivers encapsulated in the *GuiServ* cell. A task of this type determines the receiver application for each event and either delivers the corresponding event directly to the application or transforms the event into a request associated with the application. *GuiServ* thus allows screen updates without involving any application for actions like window movements.

Window Management (WM) Task. The WM Task controls the visible regions occupied by client windows. It handles requests for window management tasks such as window creation, movement, and destruction. The WM Task manages the Screen-Region Map and related data structures that represent the application windows, and has exclusive write-read access to them (see Figure 5). This task is also implemented as a MAS and may be triggered by Input Event Handler Tasks and the Front-End Task.

Controlling the exclusive access between the WM Task and the Rendering Tasks to the Screen-Region Map and related data structures is an important aspect in *GuiServ*'s design, since it can directly affect the responsiveness of *GuiServ*. We are currently evaluating the use of an adaptation of the Multipro-

cessor Bandwidth Inheritance (M-BWI) protocol [6], which is a synchronization protocol for shared-memory multiprocessors closely related to the MAS/M-CBS scheme [3, 4]. We are comparing with an alternative that takes advantage of two specific properties of our design: 1) the WM Task is the only that requires exclusive read-write access to the shared data structures, and 2) when a Rendering Task cannot access the shared data structures because of the WM Task, the Rendering Task can continue servicing requests and operating on its VFB.

4 Implementation Status

Our current *GuiServ* prototype is based off of the Nano-X Window System and contains most of the elements described in Section 3.2. We chose Nano-X because it is simple and of manageable size, implements a client/server model that fits well into Tessellation's service model, directly accesses the framebuffer, and provides a compatibility layer for X applications. These characteristics made it easy for us to rearchitect Nano-X to implement the software architecture proposed in Section 3.2.

Our prototype uses the MAS/M-CBS [3, 4] algorithm to schedule multiple Rendering Tasks on multiple cores and a virtual framebuffer per client. It also includes the Front-End Task, which is a periodic task scheduled under global EDF. USB device drivers for mice and keyboards, and the corresponding Input Event Handler Tasks (at user-level) are currently under development and we have started to port Nano-X's window manager to Tessellation. A small set of visual applications, including a video player and an image viewer, have been ported to Tessellation and work with *GuiServ*. We expect to augment this set rapidly once we finish implementing the client libraries to facilitate the development of visual applications.

Regarding Tessellation, the current prototype runs on both Intel x86 platforms and RAMP Gold [19], which is a FPGA-based simulator that models up to 64 in-order 1-GHz SPARC V8 cores. Our prototype implements several time-multiplexing policies for cells, including no-multiplexing, time-triggering, and best-effort policies; an implementation of the event-triggering policy is underway.

Tessellation currently offers two user-level scheduling frameworks that enable construction of application-specific schedulers. One is *Lithe* [15] for building hierarchical cooperative (non-preemptive) user-level schedulers. Lithe allows efficient hierarchical composition of parallel libraries and runtimes within Tessellation OS. The other is *Pulse* for implementing preemptive user-level schedulers. We have implemented various schedulers, including our MAS/M-CBS and Global EDF, using Pulse.

The inter-cell channels provide asynchronous and lock-free *single*-producer *single*-consumer communication at user-level, via a basic version of the Non-Blocking Buffer (NBB) [9].A preliminary implementation of the channel notification mechanism is available, but not yet integrated with the user-level scheduling frameworks.

One of the services in Tessellation is the Network Service; it is a multi-threaded implementation of the lwIP protocol stack.[3] It supports reservations and proportional share of bandwidth based on the mClock algorithm [8]. Currently, we only support Intel PRO/1000 network adapters. Aside from boot time configurations, the network driver is entirely contained in user-space allowing the Network Service to avoid having to make relatively expensive system calls when transmitting and receiving buffers. We also have primitive versions of the Policy Service and File Service that are under active development.

5 Experimental Evaluation

In this section we compare the end-to-end service time of Tessellation's *GuiServ* and Nano-X under different configurations and load conditions. We use two "versions" of Nano-X: the original Nano-X running on Linux (Nano-X/Linux) and a slightly modified Nano-X running on Tessellation (Nano-X/Tess). The latter preserves Nano-X's original design and architecture, but was modified just enough to run on Tessellation OS within a cell. Here we collectively refer to *GuiServ*, Nano-X/Linux and Nano-X/Tess as *display subsystems*.

We ran our experiments on a quad-core 3.4-GHz Intel Core i7 2600 CPU with 8MB L3 cache, 4GB RAM, and hyperthreading enabled (i.e., 8 hardware threads). It also has an integrated Intel HD Graphics 2000 device and we interact with it using the VESA standard. We use the latest version of Tessellation OS for *GuiServ* and Nano-X/Tess, and Ubuntu 11.04 (Linux 2.6.38) for Nano-X/Linux.

In these experiments *GuiServ*, Nano-X/Tess, and Nano-X/Linux all run on dedicated hardware threads. Because they are single-threaded, Nano-X/Tess and Nano-X/Linux are given only one thread to run on, but *GuiServ* is given up to 4 hardware threads (on 2 physical processors) to take advantage of its parallel architecture. To guarantee performance isolation between the window system and the clients, *GuiServ* and Nano-X/Tess are deployed in a non-multiplexed cell. We use `cpusets` to produce a similar isolated execution environment for Nano-X/Linux.

[3] `http://savannah.nongnu.org/projects/lwip`

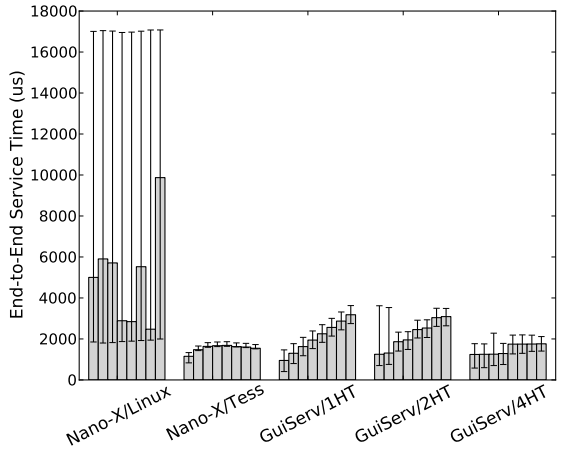

Figure 6: Average, minimum, and maximum service times for 8 video players sending inexpensive requests.

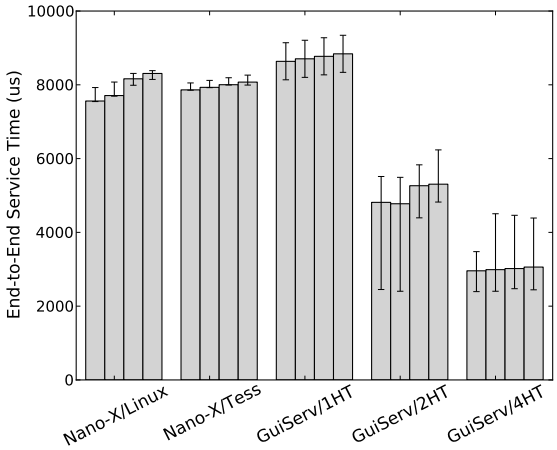

Figure 7: Average, minimum, and maximum service times for 4 video players sending expensive requests.

5.1 Experiment A

In our first experiment, we vary the number of video player clients per display system, and analyze the end-to-end service times each display system provides. End-to-end service time is defined as $t_{fb} - t_{req}$, where t_{fb} is the time at which the frame is copied to the physical framebuffer, and t_{req} is the time at which the video player sends the request with the frame to the display system.

The video format used is CIF (Common Intermediate Format); i.e., the video resolution is 352 x 288 and the frame rate is roughly 30 frames per second (fps). The frames are stored in a raw format with no compression. We purposely pick a simple raw video encoding to ensure that the video players always have the next frame ready to be sent when the presentation time comes around. Each video player is implemented as a thread of a single multi-threaded application. In this experiment, each video player sends 1000 frames at the rate of 30 fps. However, to try to stress the display subsystems, we synchronize the video players to send the frames approximately at the same time.

In Figure 6 we report the average, minimum, and maximum service times of *GuiServ*, Nano-X/Tess and Nano-X/Linux when servicing inexpensive requests from 8 video players running simultaneously. The number appended on to each *GuiServ* experiment signifies the number of hardware threads (HT) that were given to *GuiServ*. In this case, we see that Nano-X/Linux yields not only the largest average service times, but also the largest range among all the display systems. In contrast, *GuiServ* and Nano-X/Tess display stable service times with little variation. We believe this high variation in Nano-X/Linux is attributed in part to the fact that Nano-X clients write and read from UNIX domain sockets to communicate with the Nano-X server. Each of these writes and reads represent a system call into the kernel, contributing to the high variation in service times. In Tessellation, inter-cell communication is done via user-level channels. The fact that all communication is done in userspace attributes to the stable measurements we get from *GuiServ* and Nano-X/Tess.

The slope that we see in all *GuiServ* cases is due to the natural ordering in which clients send their frames. We can see that each step in the slope corresponds to however many hardware threads that are given to *GuiServ*. Normally, M-CBS would smooth out such a curve, but our M-CBS scheduler works at the granularity of milliseconds and the processing time required to serve each frame simply is not large enough for the M-CBS scheduler to step in.

We also ran the experiment with 1 client to get a baseline reading for our experiments. We saw that Nano-X/Linux had an average, minimum, and maximum service times of 271 μs, 242 μs, and 443 μs, respectively; Nano-X/Tess 255 μs, 252 μs, and 298 μs; and *GuiServ* 891 μs , 395 μs, and 1391 μs. *GuiServ* has an average service time larger than the other two because the Front-End Task in *GuiServ* polls for new requests every millisecond. This increases the average service time by 500 μs, which is the average time that requests will wait in the channel until picked up by the Front-End Task. We expect this delay to go away once we use channel notification events to activate the Rendering Tasks. There is also the scheduler overhead which explains the difference in minima between *GuiServ* and the Nano-X implementations.

In addition to serving simple draw requests, *GuiServ* should be able to service computationally-intensive rendering requests. In our case, this load was simulated by adding a constant delay (spinning) to each frame request. Figure 7 shows service times for 4 video clients, each sending computationally intensive requests, and Figure 8 shows service times for 8 video clients. In Figure 7, we see that all display subsystems

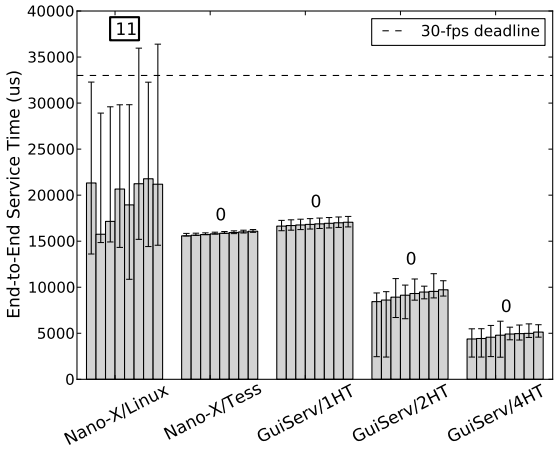

Figure 8: Average, minimum, and maximum service times for 8 video players sending expensive requests. Above each group of clients is the total number of deadlines missed for the group. Missed deadlines are boxed.

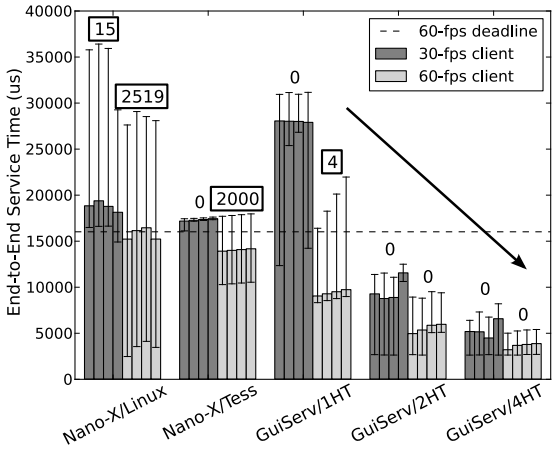

Figure 9: Service times for 30-fps and 60-fps video players sending computationally intensive requests. Above each group of clients is the total number of deadlines missed for the group. Missed deadlines are boxed.

have similar service times when running on one hardware thread. However, for an overloaded system with 8 video clients (Figure 8), Nano-X/Linux shows large variability, while *GuiServ* and Nano-X/Tess show stable service times. In addition, we can clearly see the effects of parallelization in *GuiServ*. In both Figures 7 and 8, the average service time for each client is halved every time *GuiServ* receives twice as many hardware threads. This suggests that *GuiServ* is able to efficiently use the given compute resources with little scheduling overhead.

5.2 Experiment B

In this experiment, we evaluate *GuiServ*'s capabilities to provide performance isolation and guaranteed service times to visual applications. To demonstrate this, we have 8 video players sending computationally intensive requests. This time, however, we have 4 clients make these requests for a 30-fps video, while the other 4 clients submit requests for a 60-fps video. Our goal is to show that we can adjust *GuiServ*'s CPU bandwidth per client to meet the deadlines.

The results are presented in Figure 9. Recall that each client sends 1000 frame requests. For the 30-fps video players the deadline is $\sim 33ms$, and $\sim 16ms$ for the 60-fps video players. In both Nano-X/Linux and Nano-X/Tess, we can see that the number of deadlines missed for the 60 fps video clients is significant (at least 50%). However, for *GuiServ*, by redistributing the CPU bandwidth from the 30-fps video players to the 60-fps video players, we can reduce the number of missed deadlines significantly, to almost no missed deadlines. If we increase the number of hardware threads *GuiServ* is given, we have no problem meeting all the deadlines for all the clients.

6 Related Work

To date, there have been very few attempts to improve window systems to provide response-time guarantees for visual applications. ARTIFACT [18], developed in the early 90's, is the first attempt and focuses on streaming multimedia applications. Similar in design to our *GuiServ*, the ARTIFACT server has exclusive control over the framebuffer, keyboard and mouse, and applications communicate with the server via message passing. An important distinction between ARTIFACT and *GuiServ* is how server tasks are scheduled. Tessellation's GUI Service assigns a task per client application and uses a reservation scheme to offer performance isolation among those tasks. Although the authors of ARTIFACT suggest the idea of having one task per client, they use a fixed-priority scheme, in which the priorities of the tasks are derived from the client applications, without regard for deadlines.

DOpE [7] is another proposal for a window system. Unlike our *GuiServ*, DOpE fundamentally relies on shared memory. Each client application and the window server share a compact description of the application's graphical representation. The window server can, thus, redraw the graphical representation of any client application without the active cooperation of the application.

More recently, Manica et al. [13] modified the X11 window system to achieve a similar objective. Their solution is based on the Constant Bandwidth Server (CBS) [1], which is a resource reservation algorithm. Introducing limited changes to X11 and easy portability across different X versions are some of their goals.

In contrast to the previous works, Tessellation's *GuiServ* has been designed to provide not only service-time guarantees to visual applications but also improved response times through task parallelism. For

this reason, the GUI Service uses *Multiprocessor Aperiodic Server* (MAS) [4], an adaptation of CBS reservation scheme for multiprocessor systems.

7 Conclusion and Future Work

In this paper, we showed how to construct a parallel GUI Service which provides soft real-time guarantees to visual applications. This service, called *GuiServ*, was constructed on top of Tessellation OS, a general-purpose multicore OS providing resource guarantees to applications. As demonstrated in Section 5, our service architecture scales and imposes little scheduler overhead. More importantly, we provide fewer dropped frames than a more traditional service architecture (Nano-X) running on top of Linux. As an interesting aside, *GuiServ* is now more time predictable than the original Nano-X.

In the future, we wish to augment *GuiServ* with the ability to have multiple Rendering Tasks per client. We could easily imagine the scenario in which a video player has frames it wishes to render and also a pop-up menu it wishes to draw. In this scenario, the low-latency menu draw request should not be blocked by the computationally intensive frame rendering requests. One option would be to allocate another Rendering Task for the client so that it can start drawing the menu on to the VFB. The obvious disadvantage is that synchronization will be required among Rendering Tasks and this may affect resource reservations.

We will add GPU support to *GuiServ*; the goal is to have *GuiServ* arbitrate access to the GPU for QoS enforcement. In addition, we are working on hardware acceleration mechanisms for inter-cell message communication to reduce the overall service time for requests.

We also plan to investigate how we can apply adaptive resource allocation policies to *GuiServ*. The service should support graceful degradation, in the case it loses some of its resources, and should be able to scale to utilize any extra resources it may receive. We are also looking at managing resources other than just CPU time; for instance, we are investigating a memory management and page replacement policy tailored specifically for *GuiServ*.

We are applying *GuiServ*'s software architecture and the lessons learned from this work to the design of Tessellation's Network and File Services (among others). The Network Service, for example, assigns a protocol-stack processing task per client application, and uses a reservation scheme to offer performance isolation among the tasks. Our goal, as with this work, is to reduce service times by parallelizing heavy-lifting tasks and to provide soft real-time guarantees.

References

[1] L. Abeni and G. Buttazzo. Resource reservations in dynamic real-time systems. *Real-Time Systems*, 27(2):123–165, 2004.

[2] P. Barham et al. Xen and the art of virtualization. In *Proc. of SOSP'03*, pages 164–177.

[3] S. Baruah, J. Goosens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessors. In *Proc. of RTAS'02*, pages 154–163.

[4] S. Baruah and G. Lipari. Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation. In *Proc. of ECRTS'04*, pages 109–116.

[5] J. A. Colmenares et al. Resource management in the Tessellation manycore OS. In *Proc. of HotPar'10*.

[6] D. Faggioli et al. The multiprocessor bandwidth inheritance protocol. In *Proc. of ECRTS 2010*, pages 90–99.

[7] N. Feske and H. Hartig. DOpE – a window server for real-time and embedded systems. Technical Report TUD FI03 10 September 2003.

[8] A. Gulati et al. mClock: handling throughput variability for hypervisor IO scheduling. In *Proc. of OSDI'10*, pages 1–7.

[9] K. H. Kim et al. Efficient adaptations of the non-blocking buffer for event message communication between real-time threads. In *Proc. of ISORC'07*, pages 29–40.

[10] J. W. Lee et al. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proc. of ISCA'08*, pages 89–100.

[11] J. Liedtke. On micro-kernel construction. *ACM SIGOPS Oper. Syst. Rev.*, 29:237–250, 1995.

[12] L. Luo and M.-Y. Zhu. Partitioning based operating system: a formal model. *ACM SIGOPS Oper. Syst. Rev.*, 37(3):23–35, 2003.

[13] N. Manica et al. QoS support in the X11 window system. In *Proc. of RTAS'08*, pages 103–112.

[14] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of ICDCS'82*.

[15] H. Pan et al. Composing parallel software efficiently with Lithe. In *Proc. of PLDI'10*, pages 376–387.

[16] M. Paolieri et al. Hardware support for WCET analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37:57–68, June 2009.

[17] J. Rushby. Partitioning for avionics architectures: requirements, mechanisms, and assurance. Technical Report CR-1999-209347, NASA Langley Research Center, June 1999.

[18] J. E. Sasinowski and J. K. Strosnider. ARTIFACT: A platform for evaluating real-time window system designs. In *Proc. of RTSS'95*, pages 342–352.

[19] Z. Tan et al. A case for FAME: FPGA architecture model execution. *SIGARCH Comput. Archit. News*, 38(3):290–301, 2010.