Stacked-widget Visualization of Scheduling-based Algorithms

Tony Bernardin Brian C. Budge Bernd Hamann
Institute for Data Analysis and Visualization (IDAV)
Department of Computer Science, University of California, Davis
One Shields Avenue, Davis, California
{tbernardin,bcbudge,bhamann}@ucdavis.edu

Abstract

We present a visualization system to assist designers of scheduling-based multi-threaded out-of-core algorithms. Our system facilitates the understanding and improving of the algorithm through a stack of visual widgets that effectively correlate the out-of-core system state with scheduling decisions. The stack presents an increasing refinement in the scope of both time and abstraction level; at the top of the stack, the evolution of a derived efficiency measure is shown for the scope of the entire out-of-core system execution and at the bottom the details of a single scheduling decision are displayed. The stack provides much more than a temporal zoom-effect as each widget presents a different view of the scheduling decision data, presenting distinct aspects of the out-of-core system state as well as correlating them with the neighboring widgets in the stack. This approach allows designers to to better understand and more effectively react to problems in scheduling or algorithm design.

As a case study we consider a global illumination renderer and show how visualization of the scheduling behavior has led to key improvements of the renderer's performance.

CR Categories: D.2.5.b [Software/Software Engineering]: Software Engineering—Testing and DebuggingDebugging aids; D.4.8.d [Software/Software Engineering]: Operating Systems—PerformanceOperational analysis; I.6.9.f [Simulation, Modeling, and Visualization]: Visualization—Visualization techniques and methodologies

Keywords: data visualization, task scheduling, out-of-core management, information visualization

1 Introduction

Software requires a high level of understanding on the part of the designer in order to ensure good system performance. Often understanding from an algorithmic level to a software component level to hot-spot inner loops are necessary to fully optimize an application. In many cases, bottlenecks can be identified by directly examining algorithms used, or by employing software profilers. The complexity of multi-threaded, and, in particular, non-deterministic software, makes understanding program flow a more difficult problem; a problem which can lead to suboptimal software performance.

One such complex software would be a system which performs task scheduling and data management. Specifically, the system schedules tasks in an environment where they can be executed on hybrid processing resources, namely CPUs or GPUs. The data referenced by the tasks is out-of-core, meaning much larger than the main memory of the host computer system. The scheduler has to ensure that data is local to the scheduling processor, moving it from disk to system RAM, or from system RAM to video RAM when necessary. When a processor requests a new task to be scheduled, the scheduler utilizes information about the location of the data, as well as task suitability to the scheduling processor as part of a heuristic for prioritizing tasks.

In this complex scheduling system, characterizing the overall system behavior is at least as important as optimizing individual components. A first approach would be to instrument the software to record global measures such as execution wall time. To retain readability such information needs to be kept fairly simple if it is to be considered directly by designers. A more successful approach has been to record much more data than could be directly understood and to use visualization to post-process it into more accessible forms [Zernik et al. 1992; Sawant 2007; Summers et al. 2004; Cox et al. 2005; Hummel et al. 1997; Reilly 1990]. A good visualization for our scheduling system would provide insight into the data flow, in particular when data movement (e.g., disk reads) cause bottlenecks; the effect of one scheduling decision on future ones; side effects due to system design choices; etc.

Smith and Munro [2002] proposed visualization techniques to illustrate the distribution of the dynamic load or usage at the level of Java classes. Their method provides snapshots of the state at distinct points in times. Deelen et al. [2007] developed a similar approach to the problem, visualizing the system's evolution in time. Users can use the time lines to specify a time-frame for which to evaluate a graph depicting the system load. Both techniques focus on the more structural aspect of Java programs in illustrating how the components are connected to one another. Taking the time-line visualization a step further, Moreta and Telea [2007] presented advanced rendering algorithms that allow for proper display of complex time-lines that properly resolve sub-pixel spans. Their visualization focuses on the understanding of dynamic memory allocations on small embedded systems. Mu et al. [2003] tackled the problem of memory utilization on cluster systems using shared-memory NUMA architectures. They presented time-line-like visualizations spanning multiple views both 2D and 3D. Their 3D views provide an additional dimension to classify the time evolution into phases delimited by parallel synchronization primitives. In their survey of requirements for effective software visualization Kienle and Muller [2007] supported the use of such multiple related views.

Overwhelmingly, prior work seems to provide visualization paradigms tuned for the analysis of single layers in the software hierarchy, in particular finer-grained system layers (parallel-loops, memory page accesses, etc.). The applicability of these types of approaches to our complex scheduler was limited. This led to the design of *Lumière*, a visualization system for detailed exploration of non-deterministic scheduling systems. We propose a novel syn-

chronized stack-based-widget approach that leverages the utility of multiple related views to provide insight across the many layers of the analyzed software. Individual widgets display several aspects of task scheduling in the context of both wide and narrow scopes in time. The information presented by each widget is synchronized to keep it relevant to the user exploration and is partially overlapped with the one shown by neighboring widgets in the stack to allow for easy transitioning. The details of the approach are discussed in Section 3.

We have applied *Lumière* to an out-of-core global illumination renderer capable of utilizing both the CPUs and the GPUs for rendering tasks, which include ray tracing, shading, shadows, and reflection. Most components are hybrid, and can be executed on either CPU or GPU, while the main decision making component only runs on the CPU. For this application, scheduling was designed for maximum throughput, and latency for the processing of tasks was considered unimportant unless it adversely affects throughput. The rendering application serves two purposes: facilitating the exposition of the scheduling system (Section 2), and evaluating the efficacy of the visualization for understanding and improving the performance of an application (Section 4).

2 Out-of-core Algorithms using Hybrid Resources

The hybrid scheduling system which forms the central backbone of our out-of-core applications was designed with a specific subset of problems in mind. It understands algorithms that can mapped to three key concepts: *kernels* which encapsulate the processing logic to complete a task; *static data* which provides the main application data base; and *transient data* which describes temporary scratchpad data. For all components the scheduling system is agnostic to the specific content (for data) or function (for kernels) they describe. The scheduling system is sketched in Figure 1.

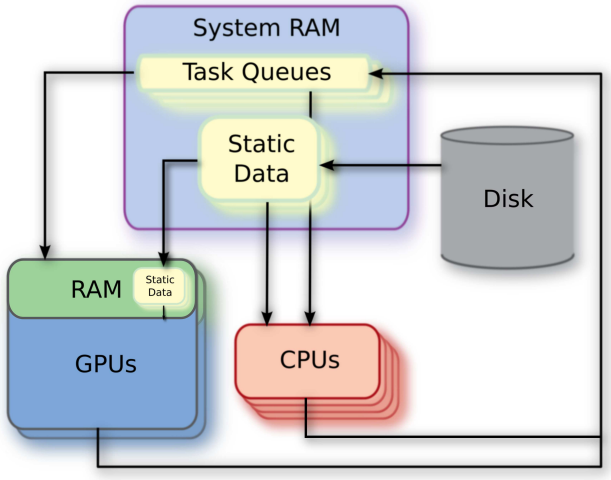

Figure 1: Data flow in our scheduling system. When work is scheduled, the scheduling system ensures that static data is in the correct location in the hierarchy for execution, and that the transient data is made available to the scheduling processor.

2.1 Data Management

The scheduling system assumes that algorithms use a large amount of static support data. This data needs to be out-of-core. It is typically split into chunks sized such that loading them from mass storage to memory as well as accessing the contained information can be done efficiently. All other data seen by the scheduling system is considered *transient* data, and is considered in-core, meaning that the data is assumed to reside in the system's main memory or in dedicated device memory. This does not limit applications from utilizing other out-of-core paradigms, however their specific kernels are responsible for managing any such data. The transient data is typically viewed by the system as defining the workload as input and then output of kernels.

The scheduling system only sees the data as numerically labeled raw memory buffers with a size and number of elements. Its responsibility, when a task must be executed, is to make sure that the referenced data is made available to the appropriate processor. In the case of transient data, for task executing on the CPU, it is only necessary to pass the data location to the scheduled kernel. If the task is to be executed on the GPU, however, the scheduling system copies the transient data to the GPU, gets the corresponding GPU memory address, and passes that to the kernel. This is done similarly for static data, with the difference that it might need to be read from disk to main memory for a CPU schedule. Because we maintain a two-level cache, if the data is already in system main memory and needs to go to the GPU, the disk read is avoided. Likewise, if the data is already on the GPU, we avoid expensive data transfer completely with respect to static data.

2.2 Task Execution

A task corresponds to an executing kernel as well as corresponding input transient data and supporting static data. The result of processing a task is potentially new transient data to be consumed by further kernels. A set of task queues implement the inter-kernel communication by serving as named repositories for transient data. They also bind the stored input to the corresponding static data chunk. The job of the scheduling system is essentially to select the most appropriate task queue when servicing an execution unit.

2.3 Recording Scheduling Decisions

Our visualization system is geared toward investigating and understanding the system state as the program progresses through its execution. In order to support this, the scheduling system can be configured to write out packets describing system state during each scheduling decision. These packets include information about the work pending in each task queue, the static chunks which are cached in various levels of the hierarchy, the processor requesting the scheduling, the time of the decision, and information about the scheduling decision taken along with a list of all possible choices.

2.4 A Rendering Application

The application that initially motivated us to develop and refine our visualization tool was a computer graphics application; specifically a photo-realistic image synthesis program based on path tracing. We were challenged to design a highly efficient approach based on scheduling. The foundational operations of path tracing are tracing rays, sampling light sources, shading, and computing reflections and refractions. The algorithm is not be discussed in detail here and we refer the reader to literature on ray tracing, path tracing, and global illumination [Dutre et al. 2002; Shirley and Morley 2003; Pharr and Humphreys 2004].

The algorithm is broken into several kernels which execute separate types of tasks. These include ray tracing, shadow tracing, light source sampling, a main decision-making kernel, and several shading kernels. Each kernel typically has one or several task

queues for transient data pending execution. The path tracer understands transient data in the form of rays, hit points, light sample requests, shaded points, and other algorithm-specific elements. The static data managed by the scheduler amounts to scene geometry pre-processed into acceleration structures for ray tracing [Wald and Havran 2006], and texture, color, and surface normal modulation information for shading and reflection.

3 *Lumière* Stack Widgets

Lumière is a tool that makes possible the offline inspection of system behavior recorded throughout the execution of multi-threaded scheduling systems. Designers of such systems require highly detailed information; for each scheduling decision, it is necessary to be able to inspect all available choices for a full characterization, which can provide debugging information when it is suspected that a flawed scheduling decision was made. It is not possible to use such detailed information alone in the context of a long execution times: many scheduled algorithms can generate several million scheduling decisions. Thus, developers need to be able to restrict their search to select times during execution where problematic behavior might be occurring.

Lumière provides layers in increasing scopes of time and coarseness of information in order to overcome this issue. Each widget presents a specialized view of the data highlighting a specific aspect of the execution, and each widget relates to other widgets providing enough redundancy for intuitive navigation of the data. The widgets can also each be manipulated to adjust the view (e.g., panning or zooming), or to highlight specific information. More specifically, selections are restricted to picking a particular scheduling decision, or one of the alternative choices for that decision; setting a specific point in time, or defining a time frame of interest. These actions are understood by all widgets and allow us to synchronize their display.

Figure 2 presents the tool in its entirety, showing all of the widgets available. The lowest level of the stack, the *Detail* view, shows the individual scheduling details for a specific scheduling event. The *Schedule* view is the next level of the stack, and it shows a progression of scheduling events over time within a small, but adjustable, time window. Level 3 of the stack shows a view of the *Workload* at a scheduling event. Level 4 is a dual-layered widget showing the *Evolution* of workload over time, where the bottom depicts a zoomed-in portion of the top. This zoom-box more readily shows the details in regions where the frequency is too high for accurate representation in the top layer. Finally, level 5 of the stack presents a *Performance* measure that graphs a metric corresponding to the "goodness" of our scheduling ability over time. Throughout the tool, color schemes are consistent to allow fast visual association.

3.1 Level 1: Scheduling Decision Details

The lowest level of the stack is the *Detail* view. It reports the raw data recorded for each decision. On the left side, a set of buttons display all of the possible tasks that can be run, each labeled with a unique identifier corresponding to "kernel:queue." The label is presented in bold face for the currently highlighted choice. To facilitate identification, buttons are also colored based on the identifier of the kernel. The layout reflects scheduling prioritization by arranging choices by decreasing priority left to right and top to bottom. When a button is clicked, the corresponding components in the other stack views are highlighted, and the details of this task are shown in the "details" panel of the widget.

The three panels on the right side of the widget show the memory state of the system at the time of the scheduling decision. This state considers the location of the static data chunks in the memory hierarchy. Buttons, colored and labeled by the chunk identifiers, are placed in one or more of the three panels depending on the residence status in main memory; the residence status in the specialized device memory (e.g., video RAM), or if the chunk is currently being read from disk.

3.2 Level 2: Schedule Timeline

The scheduling timeline is a natural progression from inspecting a single decision. It acts as an explorer for the decisions as they happen during the run of the application, and allows visualization of the system's concurrency. A timeline is generated for each CPU and GPU processing resource, as well as for a pseudo-execution unit whose job is to read the required data from disk. The events in the timelines are colored with respect to the kernels scheduled or an idle marker when no tasks could be assigned.

In order to better identify sources of latencies, a set of visual cues denotes various data transfers: clear blocks indicate that the resource is waiting on an asynchronous load from storage; quarter-sized blocks are drawn for a static data transfer from system RAM to special device RAM; and two-third-sized blocks define transfers of transient data from system RAM to device RAM.

The timeline can be interactively panned and zoomed, and clicking on a particular block selects the corresponding scheduling decision in the *Detail* view. The selected decision is highlighted with a stippled dark-red border in order to maintain a visual connection with the other components.

3.3 Level 3: Workload Detail

The *Workload* widget provides an intuitive transition between the decision-oriented bottom widgets and the workload-oriented widgets at levels 4 and 5. It does this by combining the task choices and the residence status of corresponding static data for the selected scheduling decision. The horizontal subdivisions of the widget are associated with static chunks. A color mark at the bottom of each subdivision tags the chunk. The arrangement is left to right in order of the chunk's numeric identifier. Vertical subdivisions within each horizontal section represent the various tasks requiring the associated chunk. The tasks are layered bottom to top in prioritized order, and the thickness of each layer denotes the quantity of work for the task. Finally, a color-coded highlight is drawn around the chunks' color mark. It indicates the location of the chunk in the memory hierarchy. No highlight means that the data is on mass storage, red denotes main memory, green implies the device being scheduled, blue indicates that the chunk is resident on a device other than the one being scheduled, and the RGB color combinations represent corresponding multiple locations.

The representation depicts the distribution of the task workload across the data used by the application. It supports identifying flaws in prioritization (e.g., when thicker layers appear above thinner ones), as well as flaws with respect to loading of inappropriate chunks. Users can click on any of the displayed layers to select that specific scheduling choice from the array of available ones during the current scheduling event. The selection is denoted by the same dark-red stippled line as in the schedule view and synchronized across widgets.

3.4 Level 4: Workload Evolution

The goal of the *Evolution* widget is to provide an evolving view of the workload over the course of execution. The workload corresponding to each static data chunk is color-coded as in the mark

Figure 2: The stack of widgets of the *Lumière* tool. The layout and interconnection of the widgets allow users to drill down to problem spots in the execution. At the top, *Performance* and *Evolution* views provide much needed context and coarse-grained viewing of a derived performance measure contrasting it with the actual workload distribution over the referenced static data. The *Workload* view breaks down this workload for a specific scheduling decision into its individual tasks. The scheduled tasks are shown in the *Schedule* view below, arranged in time lines for each execution unit and the raw measured details are presented in the bottom most *Detail* view. The multiple views of the synchronized state across the various layouts provide the basis for *Lumière's* effectiveness.

from the *Workload* view. The areas are stacked from bottom to top respective to the sequence of the chunk numeric identifiers.

Overview The top part of the widget shows a static overview of the evolution of workload over the entire execution time. The purpose is to hint at problem areas at a glance and to provide temporal context. Users can manipulate a rectangular cut-out window by changing its position and size. This window provides a time scope for the timelines of the *Schedule* view and a zoom-in which is displayed in the lower half of the widget. Sometimes the resolution of events is high enough that the *Schedule* view needs to be zoomed-in further. Thus, manipulation of that widget does not provide feedback to the *Workload* widget. The exception is that the selection of a decision in a timeline adjusts the position of the zoom-in window, but does not resize it.

Zoom-in The zoom-in view of the widget is useful due of the high density of workload information presented in the overall view. The zoom-in helps to avoid clutter, and allows investigation of possible overarching issues. Users can click within the workspace to select a schedule choice based on the time at the clicked location. If the position is outside the top-most area (the gray space), the choice which was actually scheduled is selected. On the other hand, if one of the colored areas is selected, the corresponding choice is selected instead.

A cross-hair shows the picked location, and a highlight is placed around the corresponding tasks workload evolution. The highlight is useful for showcasing where work is generated and consumed for that static data chunk.

3.5 Level 5: Performance Measure

At the highest level, at the scope of a whole program execution, a performance metric is graphed. It presents information derived from the recorded data rather than a display of the raw data. The purpose of the *Performance* widget is to provide hints to locations where flawed decisions or bad system behavior might be exhibited. This performance metric is application-specific and would likely need to be adapted to algorithms other than path tracing.

For our application, we know that disk access is the main bottleneck, but access patterns are difficult to predict or even detect. The scheduler ideally should maximize reuse of static data chunks. To reflect this, the metric for our performance measure is based on data residence. It evaluates roughly how well the resident set of chunks covers the workload at each decision-making event. The actual metric is based on accumulating all task workloads for corresponding static chunks, and then weighting them by the locations of those chunks. The chosen weights are shown in Table 1.

A cursor in the form of a vertical line indicates the location in time for the current scheduling decision, and users can click the view to select a new scheduling time.

unit	storage	reading	main memory	current GPU	other GPU
CPU	0.0	1.250	2.0	N/A	N/A
GPU	0.0	1.125	1.5	2.0	1.125

Table 1: Weights associated with the cumulative task units of a static chunk for the *residence* measure. A weight is zero if the data is on persistent storage.

4 Illuminating an Out-of-core Renderer

This section presents several examples of how *Lumière* was used to analyze our out-of-core rendering system (Section 2). We describe the process of identifying problems and discuss how some of the issues were resolved. Our test cases span three scenes which have 3, 17, and 33 static data chunks of roughly 250 MB each. The captured scheduling decisions number between 80 thousand to 2.3 million decisions per execution lifetime.

4.1 Honing in on Scheduling Inefficiencies

A methodology we successfully used when searching for performance issues in the scheduling system is that of "honing in" on problems. We started at a high-level view, and worked our way toward the details. Figure 3 illustrates this process for discovering shortcomings in our lazy data fetching strategy.

After loading the recorded data into the tool, we followed the progression of the performance metric (green line) in the *Performance* widget, looking for locations where the metric indicated low efficiency. At one such location, the *Evolution* view indicated that a large amount of work was actually available, which should have been a favorable case for efficiency.

To investigate, we created a zoom-box in the *Evolution* view and selected a scheduling time point within that frame. This updated the *Workload* view to provide the pending work's distribution into its different tasks. This view revealed that although a large amount of work was available it was mostly associated with chunks that were not resident in the memory caches. Instead the resident chunks had very few associated pending tasks, explaining the observed dip in the performance metric. Scheduling decisions would require the transfer of data from mass storage.

At the same time we noted that there were segments of idle-time in the *reader* process' timeline (the central row in the *Schedule* view). This insight into the *reader's* underutilization suggested that a form of data prefetching would be appropriate at that point of the renderer's execution.

Further investigation of the *Workload* view revealed that the next task picked to be scheduled was not associated with the static data chunk having the largest amount of pending work. In Figure 3 the eighth subdivision from the left denoted that chunk, however, a task was scheduled from the one shown in the seventh subdivision from the right. This happened because the scheduler only considers individual task when prioritizing. Instead, our investigation suggested that considering collections of tasks associated with the same static data could further minimize slow transfers from mass storage.

4.2 Windowed Dispatch of Rendering Tasks

Figure 4 shows the *Evolution* and *Schedule* views relevant to this example. During visualization of several executions of the renderer our attention was drawn to the distinct saw-tooth pattern of the workload displayed in the global *Evolution* view. The very low workload between the spikes was also disconcerting. In order to shed some light on the situation, we used the zoomed-in view to explore the decisions leading up to a spike.

The *Schedule* view showed that the eye-ray-dispatching kernel was being run slightly ahead of each spike. The spikes were consistent with the properties of the dispatcher, requiring only little input transient data but producing a large amount of output. Specifically, this dispatcher generated eye rays in windows of some width and height in pixels. Moreover it required windows to be processed entirely, before they could be redispatched. Because path tracing is

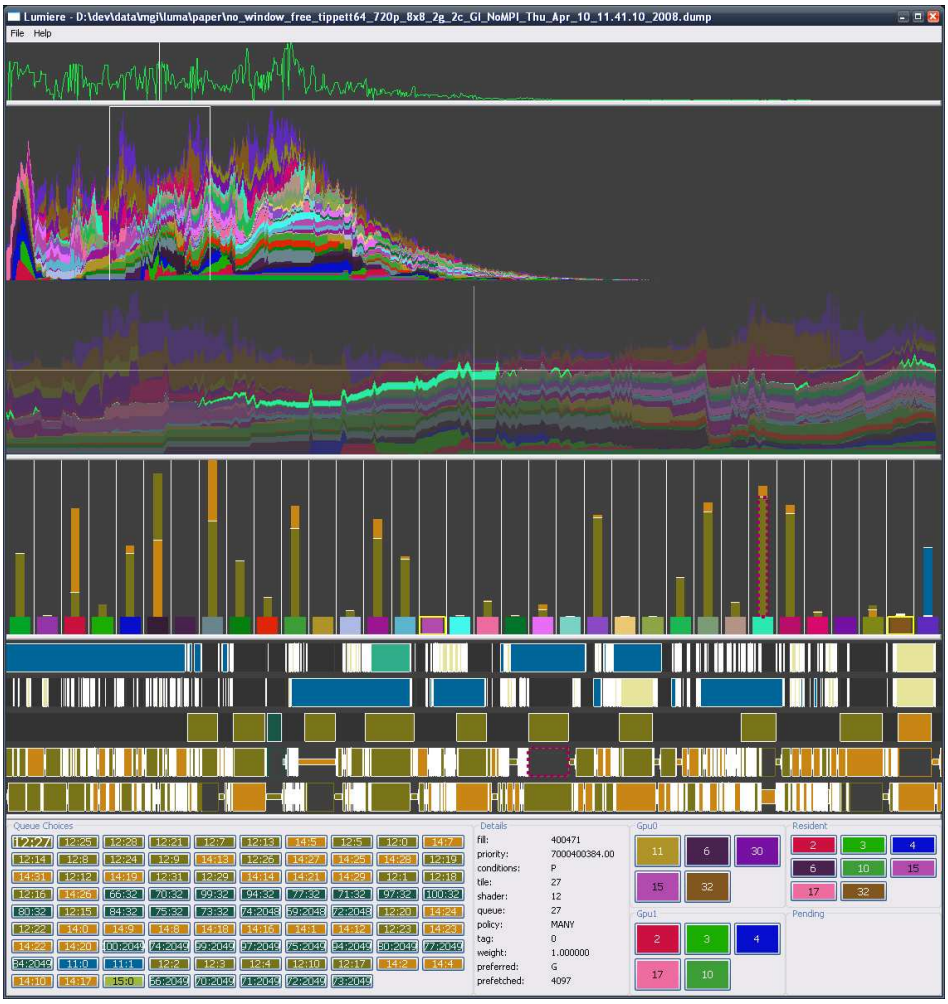

Figure 3: Using *Lumière* to find flaws in a rendering algorithm. Where the *Performance* view is indicating a dip in its measure, the *Evolution* and *Workload* views show the presence of a significant amount of tasks, but they do not have resident static data. This system is not exploiting the idle time of the static data reader (middle scheduling timeline) to improve residency of data for posted tasks.

a stochastic process, it was possible for a window to take a long time to complete, even when the majority of its pixels had been processed.

This led us to a design where we could dispatch more work each time a single pixel finished processing (with some performance trade-offs). The result is shown in Figure 5. The overall workload is noticeably higher. The graph still exhibits some spikes that can be attributed to the delayed execution of the dispatcher due to task buffering. The performance improvement from this optimization was quite dramatic: The rendering time, which was originally 2485 seconds, was reduced to 1290 seconds.

Additionally, the change allowed us to maintain more pixels in flight at a time, leading to even greater overall efficiency. The result is a graph similar to the one in Figure 3, and a further decrease of the run time to 510 seconds, corresponding to an improvement by a factor of nearly five. The spikes can still be seen in the figure, but they are severely muted due to the large amount of additional concurrent work.

The evolution of the workload with the above optimization produced a new issue that is clearly displayed in the *Evolution* view of Figure 3: The execution terminates in a trail where considerably less work is processed but a large amount of time is spent processing it. To some extent this behavior is actually to be expected of the path tracing algorithm we used. In fact, Figure 5 also exhibits it. Comparing the two figures provides us with some insight into the nature of our optimization and the limitations of the chosen algorithm. Recall that the overall rendering time was shortened from 1290 seconds in the case of Figure 5 to 510 seconds for Figure 3. In both cases the trail actually took roughly the same amount of processing time but the preceeding work was accelerated going from the former to the latter.

4.3 Lock Contention on Task Queues

This example illustrates the effectiveness of the *Schedule* views in highlighting problematic behavior by exposing patterns. The motivation for the sequence of visualizations came from observing poor scalability of the renderer with respect to increases in workload. This behavior was conflicting with the core design of the scheduling system. In fact, the issue was so prominent that it appeared even when the entire static data of the scene could fit in-core.

The top of Figure 6 shows our first discovery: Transfers of relatively small (maximum of 50 MB) transient data to GPU required

Figure 4: *Evolution* and *Schedule* views when using windows to dispatch pixel computation tasks. *Schedule* view, from top to bottom: two CPUs, one *reader* and two GPUs. Before each "spike" the dispatcher kernel is executed (light-colored box with the dark, dashed outline in the first CPU's timeline).

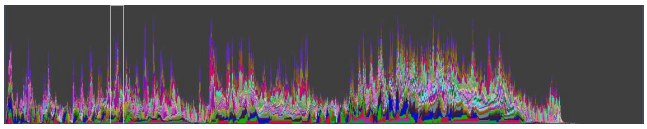

Figure 5: *Evolution* view for a run using a pixel-based dispatcher. "Spikes" in the workload are still present, however noticeably more work is present between them.

disproportionately long time when compared with the actual execution of a kernel or the transfer of the much larger ($\approx$ 250 MB) static data chunks both from disk, and across the PCI-Express bus. For context, the small, thin vertical bar on the third timeline represents a disk read of static data, usually the slowest process in our system.

The transient data in question was produced by our dispatching kernel running on the CPUs (the top-most two light-colored segments). As the ray intersection kernel was the only other running kernel (bottom timeline), we suspected that these two kernels were adversely affecting each other in their producer/consumer interaction via shared task queues.

A second experiment was performed with the GPU disabled, and two additional CPUs for a total of four. A cut-out of the *Schedule* view (bottom of Figure 6) shows a honey-comb-like pattern consistent throughout the execution. The regular offsets in the schedule times reinforced our intuition about the system behavior. As a result, we optimized the task queue data structure to reduce the potential for lock contention on this centralized resource. This led to a reduction in runtime from 700 to 113 seconds for the test scene. An advanced software profiler likely could have directed us to the same conclusion for this issue. The example shows that even some "low-level" issues can be effectively found using our visualization framework.

4.4 Error Prioritizing Chunk Residence

Although our rendering application is typically executed with both CPUs and GPUs, it was also necessary to benchmark the system with only CPUs. Some of these experiments provided cause for

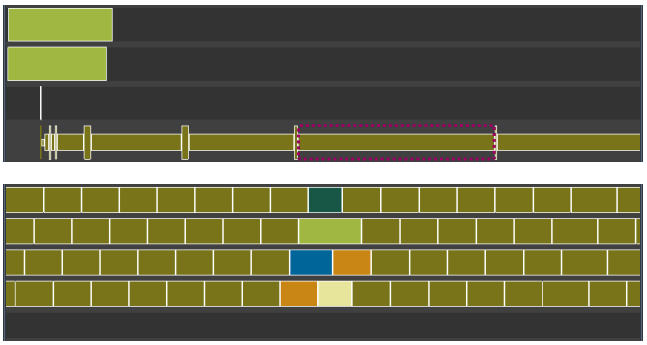

Figure 6: *Schedule* views showing, top: the initial phase of a rendering run with two CPUs, one reader and one GPU (top to bottom) exhibiting suspiciously long transient data transfers; bottom: a run with four CPUs exhibiting suspicious offsets in the times of execution.

concern as some execution runs would take significantly longer to finish.

Figure 7 shows a sequence of *Lumière* views that was instrumental in solving a scheduling issue causing the performance problem. It also highlights the value of being able to investigate the details of all the scheduling choices. On the left half, the *Evolution*, *Workload* and *Schedule* views are presented. In the *Evolution* view, we first noticed an obvious stagnation in the system's progress after only a short time running. We confirmed it using the *Workload* view at corresponding scheduling decisions. The display of the associated *Schedule* view further provided a telling interaction between the top two CPUs and the bottom *reader* timelines: The CPUs spent progressively less time in actual execution, and progressively more time waiting for read requests to complete. Correspondingly the *reader* thread serviced these requests uninterrupted.

The right side of Figure 7 presents two zoomed-in views of the *Schedule* view with the corresponding *Detail* view. In the top half, we selected a specific CPU read request and observed the demand for chunk 0 to be read in order to execute kernel 14. In the bottom half, the immediately following scheduling decision for that CPU was selected. We observed it to be yet another read request, this time for chunk 27. Perusing the choices, the previous task, 14:0, was still present but having a much lower priority.

The observations led to the investigation of the prioritization code for the scheduler. The system was properly registering the presence of the chunk in main memory: "Cr" meaning CPU-resident, versus "Cp" meaning CPU is pending read). However, the prioritization was mis-ordering the choices. As a consequence, the scheduling decisions would favor out-of-core chunks and continually thrash the caches, only occasionally making progress.

5 Conclusions and Future Work

We have presented a visualization tool consisting of several widgets interconnected in a stack that facilitates exploration of scheduling data. The tool was shown to be effective in improving our understanding of our scheduling and rendering software. The visualization widget stack aided in locating several implementation bugs within the scheduler, and has also helped to pinpoint conceptual problems with our implementation of the rendering algorithm.

Because *Lumière* is built to interface with a general-purpose hybrid out-of-core scheduling framework it can be directly used to visualize the behavior of any algorithm implemented on top of this

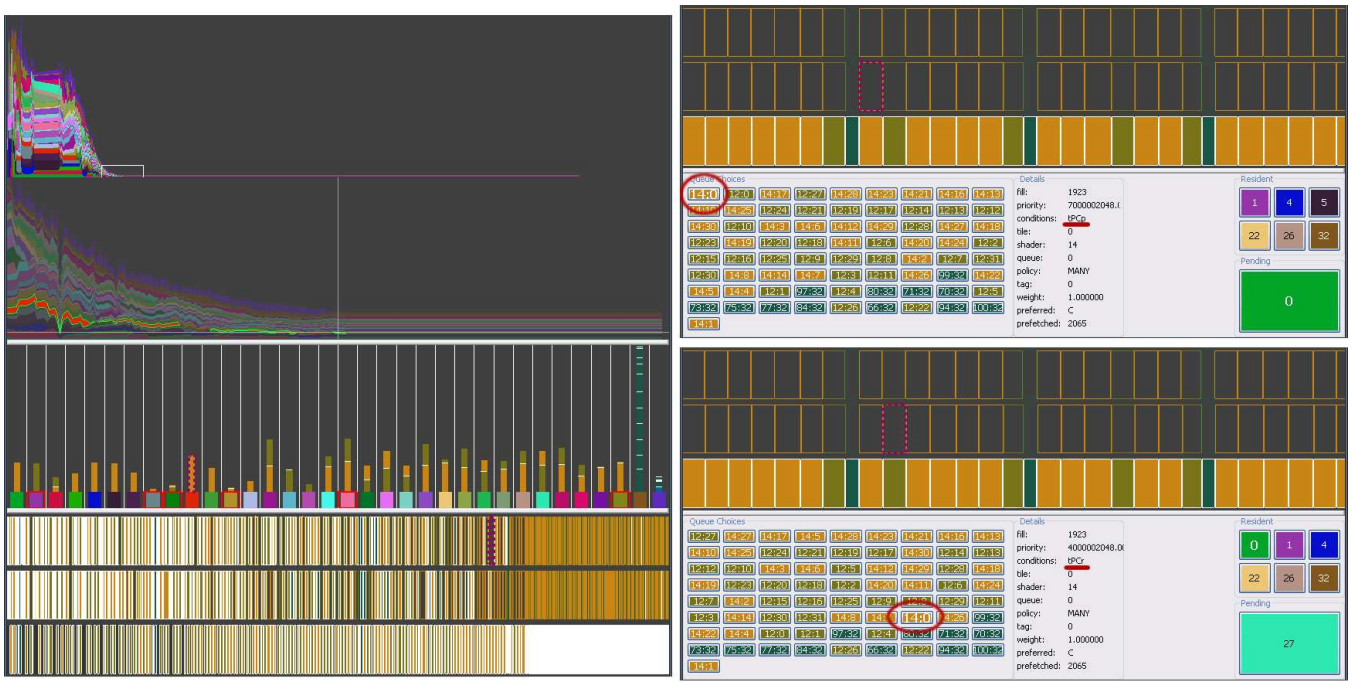

Figure 7: Discovering flaws in scheduling prioritization details. On the left: *Evolution*, *Workload* and *Schedule* (from top to bottom, two CPUs and one Reader) views displaying the stagnation of the renderer's progress. On the right: zoomed-in views of the *Schedule* and accompanying *Detail* views. For two succeeding events on the same execution unit, prioritization is erroneously favoring chunks on disk over those resident in main memory.

framework. In particular, we used it to analyze a stochastic path tracer. Other algorithms which would fit well include large Monte Carlo simulations, or particle advection systems in out-of-core flow fields. The data *Lumière* assumes to be captured from the considered application is fairly general. It is likely that most hybrid out-of-core systems could easily be modified to produce it. However, we concede that the design of the particular visualization widgets used is focused on highlighting issues related to out-of-core data transfers. Those were the bottlenecks most prominent in our rendering application.

In the future, we would like to explore the efficacy of the *Lumière* stack paradigm applied to different sets of widgets. One can imagine widgets more focused on illuminating the communication between kernels, or the evolution of the memory usage. More specific to our application, we would like to investigate better/additional performance metrics that might be more indicative of bad behavior. Some additional useful features would have been: being able to simultaneously see all times when a given chunk is resident, or all times when a specific kernel or task queue is scheduled; visualizing the durations of residency for chunks in various levels of the memory hierarchy. The timeline display of the *Schedule* view might be improved by implementing a technique similar to the cushions proposed by Moreta et al. [2007] to improve visual quality.

Currently the system has several limitations that might be resolved. While our software allows exploration of out-of-core algorithms, *Lumière* does not itself handle out-of-core data. Future implementations should allow investigating very large captured schedule-data from long software execution runs. Additionally, the software relies heavily on the user's expert knowledge of the visualized application for locating bottlenecks; *Lumière* does no analysis to automatically pinpoint specific problems. Future work could go in this direction, however, making this analysis agnostic to the underlying application may prove difficult. Likely this would lead to a visualization system with different analytic plugins for distinct applications.

Finally, adding support for comparative visualization may be helpful. Many issues only come to light when comparing several slightly adjusted runs of the application. Some examples of how *Lumière* could support this would be to overlap performance metric graphs of multiple executions, or synchronizing two visualizations to the same time scale.

Acknowledgements

We would like to acknowledge funding by the Office of Science, U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET) and by a United States Department of Education Government Assistance in Areas of National Need (DOEGAANN) grant No. P200A980307. Additionally, we would like to thank the members of the Visualization and Computer Graphics Research Group of the Institute for Data Analysis and Visualization at UC Davis.

References

COX, P., GAUVIN, S., AND RAU-CHAPLIN, A. 2005. Adding parallelism to visual data flow programs. In *Proceedings of the ACM Symposium on Software Visualization*, ACM, 135–144.

DEELEN, P., VAN HAM, F., HUIZING, C., AND VAN DE WETERING, H. 2007. Visualization of dynamic program aspects. *International Workshop on Visualizing Software for Understanding and Analysis 0*, 39–46.

DUTRE, P., BALA, K., AND BEKAERT, P. 2002. *Advanced Global Illumination*. A. K. Peters, Ltd.

HUMMEL, S. F., KIMELMAN, D., SCHONBERG, E., TENNENHOUSE, M., AND ZERNIK, D. 1997. Using program visualization for tuning parallel-loop scheduling. *IEEE Concurrency 05*, 1, 26–40.

KIENLE, H. M., AND MULLER, H. A. 2007. Requirements of software visualization tools: A literature survey. *International Workshop on Visualizing Software for Understanding and Analysis 0*, 2–9.

MORETA, S., AND TELEA, A. 2007. Visualizing dynamic memory allocations. *International Workshop on Visualizing Software for Understanding and Analysis*, 31–38.

MU, T., TAO, J., SCHULZ, M., AND MCKEE, S. A. 2003. Interactive locality optimization on numa architectures. In *Proceedings of the ACM Symposium on Software Visualization*, ACM, 133.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc.

REILLY, M. 1990. Presentation tools for performance visualization: the m31 instrumentation experience. *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences 1*, 307–313.

SAWANT, A. 2007. Diffarchviz: A tool to visualize correspondence between multiple representations of a software architecture. *International Workshop on Visualizing Software for Understanding and Analysis 0*, 121–128.

SHIRLEY, P., AND MORLEY, R. K. 2003. *Realistic Ray Tracing*. A. K. Peters, Ltd.

SMITH, M. P., AND MUNRO, M. 2002. Runtime visualisation of object oriented software. *International Workshop on Visualizing Software for Understanding and Analysis 0*, 81.

SUMMERS, K. L., CAUDELL, T. P., BERKBIGLER, K., BUSH, B., DAVIS, K., AND SMITH, S. 2004. Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers. *Information Visualization 3*, 3, 209–222.

WALD, I., AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 61–69.

ZERNIK, D., SNIR, M., AND MALKI, D. 1992. Using visualization tools to understand concurrency. *IEEE Software 9*, 3, 87–92.

Figure 2: The stack of widgets of the *Lumière* tool. The layout and interconnection of the widgets allow users to drill down to problem spots in the execution. At the top, *Performance* and *Evolution* views provide much needed context and coarse-grained viewing of a derived performance measure contrasting it with the actual workload distribution over the referenced static data. The *Workload* view breaks down this workload for a specific scheduling decision into its individual tasks. The scheduled tasks are shown in the *Schedule* view below, arranged in time lines for each execution unit and the raw measured details are presented in the bottom most *Detail* view. The multiple views of the synchronized state across the various layouts provide the basis for *Lumière's* effectiveness.