Performance Analysis and Visualization of Parallel Systems Using SimOS and Rivet: A Case Study

Robert Bosch, Chris Stolte, Gordon Stoll, Mendel Rosenblum, and Pat Hanrahan
Computer Science Department
Stanford University

Abstract

In this paper, we present an evolving system for the analysis and visualization of parallel application performance on shared memory multiprocessors. Our system couples SimOS, a complete machine simulator, with Rivet, a powerful visualization environment. This system demonstrates how visualization is necessary to realize the full power of simulation for performance analysis. We identify several features required of the visualization system, including flexibility, exploratory interaction techniques, and data aggregation schemes.

We demonstrate the effectiveness of this parallel analysis and visualization system with a case study. We developed two visualizations within Rivet to study the Argus parallel rendering library, focusing on the memory system and process scheduling activity of Argus respectively. Using these visualizations, we uncovered several unexpected interactions between Argus and the underlying operating system. The results of the analysis led to changes that greatly improved its performance and scalability. Argus had previously been unable to scale beyond 26 processors; after analysis and modification, it achieved linear speedup up to 45 processors.

1. Introduction

Computer systems are rapidly increasing in complexity. Processors contain more and more transistors and networks connect more and more computers. Unfortunately, designing and evaluating complex systems is becoming more difficult. A good example of this complexity is the development of parallel applications on a modern multiprocessor system. The programmer must not only understand the idiosyncrasies of a superscalar processor and the bandwidths and latencies of a non-uniform memory system, but also the application's interactions with the operating system and other concurrently running applications.

Performance analysis of large parallel applications has traditionally been done using instrumentation: data collection code is typically added to either the application under study or an underlying message passing library. However, there are inherent limitations to this approach to performance analysis. Performance problems on parallel systems are often timing-dependent, and the intrusive nature of instrumentation can perturb the problem being studied. Data collection is limited to the hardware and software components that are visible to the instrumentation code, potentially excluding detailed hardware information or operating system behavior.

Further, due to the complexity of large parallel systems, the performance analysis process is often an iterative one, in which data collection and analysis must be refined over several executions of the application before the problem is discovered. However, the precise timing characteristics of a program may change from one run to the next, forcing the analyst to pursue a moving target.

In recent years, complete machine simulation has become an increasingly important tool for the study of parallel systems performance and behavior. The SimOS [8] simulation environment has been used in a range of studies, and it has proven to be an effective tool for the analysis of computer systems. Several features of complete machine simulation are particularly valuable for the study of parallel applications:

- **Visibility**. SimOS provides complete access to the entire hardware state of the simulated machine: memory system traffic, the contents of the registers in the processor, etc. It also has access to the software state of the system, enabling machine events to be attributed to the processes, procedures and data structures responsible for them.
- **Flexibility**. The amount of data that can be collected by SimOS during a simulation run is potentially immense. Therefore, SimOS provides a flexible mechanism for focusing the data collection, called *annotations*. Annotations are simple Tcl scripts that are executed whenever an event of interest occurs in the simulator. These scripts have access to the entire state of the simulated machine.
- **Repeatability**. SimOS is a completely deterministic simulator: two simulations with identical initial conditions and hardware configurations will produce identical cycle-by-cycle results. This is crucial for performance analysis: an initial analysis session often suggests a more focused data collection scheme to better understand the behavior of the system. Using

SimOS, the exact run may be repeated with annotations added to provide more detail.

- **Configurability**. SimOS can be configured to model a variety of hardware configurations. This enables the study of system configurations that are not readily available. In this case study, we use SimOS to model an SGI Origin [10] multiprocessor with up to 64 processors. Since we do not have ready access to such a large-scale machine, this study would not have been otherwise possible.

While simulation addresses many of the problems of instrumentation, it also presents a data analysis challenge. The complete visibility of hardware and software structures provided by simulation can produce a huge amount of data. Most simulation studies have dealt with this challenge by using statistics and aggregation to reduce the amount of data to be analyzed [1][3][14]. However, the data reduction process can easily obscure the individual events that may be the bottleneck for an entire application. In order to unleash the full potential of simulation, we need tools that support navigation and exploration of these large, complex data sets, providing a summary while preserving the individual data elements for detailed analysis.

In this paper, we present an evolving system for the analysis and visualization of parallel application performance on shared memory machines. Our system couples SimOS with Rivet, a visualization environment that integrates high-performance data and graphics objects with a high-level scripting language, providing both flexibility and performance. The system combines the determinism provided by simulation with the flexibility of the visualization environment to create an iterative performance analysis framework.

We demonstrate the effectiveness of this system with a case study of the Argus rendering system [9], a parallel graphics rendering library targeted at large shared memory multiprocessors. Using standard instrumentation and data analysis tools, the developers of Argus were unable to scale their application beyond 26 processors. We studied Argus under simulation, developing two custom visualizations within our framework: MView, which displays several detailed views of the application's memory system behavior, and PView, which presents a variety of process attributes such as CPU scheduling and exception handling. These visualizations enabled us to discover several bottlenecks that were limiting the scalability of Argus. As a result of our analysis and visualization, we extended the linear scalability range of Argus up to 45 processors.

2. Related work

Performance analysis of large parallel applications is an area of increasing importance, and considerable research has been done on developing general analysis and visualization systems for this task. Existing systems include ParaGraph [7], Pablo [13], PARvis [12] and Paradyn [11]. All of these systems use instrumentation for data collection: ParaGraph and PARvis instrument the message passing subsystems of distributed memory multiprocessors, and Pablo and Paradyn perform instrumentation of the applications themselves. While these systems try to minimize the perturbation caused by data collection, the behavior of the applications under study is still affected by the instrumentation. We use simulation, which gives us nonintrusive access to both machine and program state.

ParaGraph [7] and PARvis [12] are both systems for visualizing the behavior and performance of parallel applications on message passing architectures. One of the strengths of these systems is their inclusion of a wide variety of visual representations for system attributes such as message traffic and processor utilization. However, it is relatively difficult to extend these systems to support new data displays or data sources. While our system presently includes fewer visual primitives, we provide a simple mechanism for designing new primitives and connecting them to data objects, which is critical when integrating visualization with simulation.

The Paradyn [11] system shares several design goals with our system. Paradyn is an extensible system, supporting the addition of new visual primitives. In order to handle the overwhelming amount of data available in the study of large-scale multiprocessors, Paradyn utilizes dynamic instrumentation. This approach allows data collection to be refined during the course of a run, enabling the user to focus on attributes of interest. While the notion of refining the data collection based on observed behavior is a powerful one, dynamic instrumentation presumes that events of interest will recur in the future. Paradyn guides the dynamic instrumentation using search algorithms, resulting in smaller data sets. Thus, they have not developed sophisticated visual representations and aggregation techniques for large data sets as we have developed in our system. Their use of computational algorithms rather than human intuition and pattern recognition results in a system with quite different goals and challenges.

The Pablo research group has developed several performance analysis tools, including Pablo [13] and SvPablo [5]. These systems have focused on presenting the statistics gathered during execution in one or two complex views, such as multi-dimensional scatterplot arrays and hierarchical source code views. In contrast, our system presents many views for correlating statistics with resources such as memory, processors, and source code. Neither Pablo nor SvPablo are easily extensible, except for the enhancement of existing views.

3. Integrating visualization with simulation

The architecture of our system, which is depicted in Figure 1, couples SimOS with Rivet. The performance

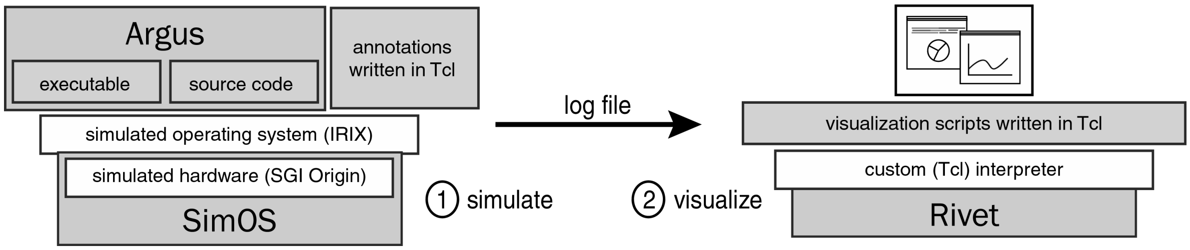


Figure 1: System architecture and data flow. The system was built to support an iterative analysis. The user first simulates an application (such as Argus) on SimOS, a complete machine simulator. Tcl scripts called annotations are used to control the data collection process. The data is then displayed using visualizations developed within Rivet. These visualizations are specified with visualization scripts, also written in Tcl. The user interacts with the visualization, gaining insight into the application's performance problems. The user may then change the application, the annotations or the visualization scripts and repeat the process.

analysis process begins by running the application to be studied on SimOS with a standard set of data collection annotations; the resulting data is stored in a log file. The data is then displayed using visualizations developed within Rivet. The user interacts with the visualization, gaining insight into the performance problems being experienced. The user may then change the application itself, the set of data to be collected, or the visualizations being used, and repeat the process.

Our performance analysis of Argus enabled us to identify several critical features required of a visualization system in order to realize the full power of simulation as a performance analysis tool:

- The flexible data collection mechanisms of simulation demand a flexible visualization system.
- The exploratory nature of performance analysis requires interactivity.
- The demands of handling large data sets and providing continuous real-time display imply a need for aggregation and rendering techniques.

We now discuss the importance of each of these elements in the integration of simulation and visualization, and describe the support provided in Rivet for these features.

3.1. Flexibility

Traditional data collection techniques used for performance analysis tend to focus on a particular aspect of the computer system, such as the message-passing interface or the memory system. Because data collection is focused on a restricted domain, a visualization system can provide a fixed set of visual representations. As discussed in Section 1, however, a complete machine simulation tool such as SimOS provides a flexible data collection mechanism. The ability to collect detailed information about nearly all aspects of the hardware and software demands much more flexibility from the visualization system. In many cases, the process of analysis and visualization may reveal either that the data collected was not pertinent or that the available data was insufficient to explain the observed behavior. In either case, the user must be able to modify or augment the data collection to retrieve the information of interest and adapt the visualization system to present this new data. Because such a wide range of data is visible to the simulator, it is not feasible to provide prebuilt representations for every data item. Rather, the system must provide mechanisms to quickly create new representations.

Rivet provides flexibility for the design of visualizations through a compositional architecture. Visualizations are composed of two basic classes of objects: data primitives and visual primitives. These primitives are written in C++ and OpenGL, and are designed for performance. By providing extensive support for features common to most visual primitives, such as color selection, layout, and the rendering of geometric primitives, text and images, Rivet greatly simplifies the task of building new primitives.

To develop an interactive visualization from these data and visual building blocks, the user writes Tcl scripts that specify: (a) how data is imported from an external data source into the data primitives, (b) which visual primitives are used to display the data, and (c) how the visual primitives interact with the user and with one another.

This combination of Tcl and C++/OpenGL enables Rivet to be used to construct high-performance, interactive visualizations in a rapid and flexible fashion. A sample script in the Appendix illustrates how easily a visualization can be constructed for data collected from SimOS.

3.2. Exploration

In addition to flexibility, the visualization system must also support exploration: the data sets generated by SimOS can be very large, and static visualizatons do not fully leverage the power of the human perceptual system to discover important patterns and relationships. In order to expose the underlying patterns and interesting events within the data, the user must be able to directly manipulate the visual representations of the data. Rivet includes two essential interaction features for supporting effective

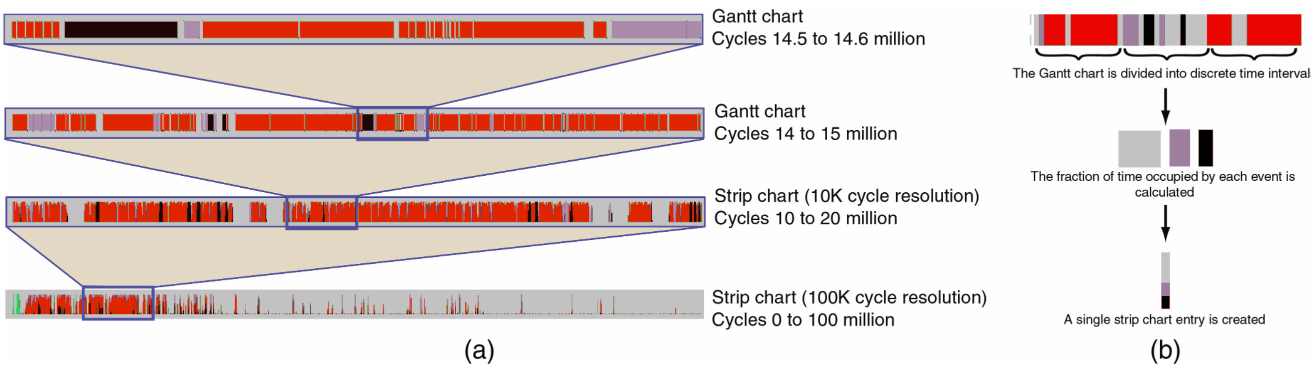


Figure 2: Argus simulation runs consist of roughly 100 million cycles of information per processor. For efficient navigation and display, we form an aggregation structure as shown in (a). We divide the execution time into discrete time slices, and for each slice we compute the fraction of time occupied by each event type as shown in (b). These aggregations are built at resolutions of 10K, 50K and 100K cycles. Depending on the number of cycles selected to be viewed and the available screen space, the appropriate level of the aggregation structure is displayed.

exploration of performance data: interactive layout and dynamic queries.

The ability to interactively modify the layout of the charts and visualization is vital in leveraging the human perceptual system to make comparisons and detect patterns, as demonstrated by Bertin [2]. The visualizations we developed for parallel performance analysis display many collections of objects such as processes, nodes, and source files, that have no single “natural” ordering or priority. These objects are initially displayed with an arbitrary ordering and pixels are uniformly allocated across all elements of the collection. However, visual exploration of the data may quickly reveal that only a subset of the entities is interesting, or that there are deeper relationships between particular members of the display. In order to support this exploratory process, the user must be able to alter the orderings and relative sizes of the charts. Several systems, including Rivet, provide mechanisms for procedurally sorting charts. This approach is useful when the user has an *a priori* understanding of the possible relationships. However, Rivet also provides support for direct manipulation of the elements in a layout. The manual rearrangement and sizing of charts allows the user to uncover patterns in the data and discover the underlying relationships in an *a posteriori* fashion.

Dynamic queries [15] are a common technique used to allow the user to quickly explore a large data set, filtering out unwanted information by adjusting controls such as sliders and buttons with continuous feedback. In the visualizations we developed for our analysis of Argus, we provide query controls for both the time dimension and for selecting which attributes of the collected data to display. By providing these interaction techniques, Rivet enables the user to explore hundreds of millions of cycles of data and rapidly isolate areas of interest.

3.3. Aggregation

The final feature of Rivet we discuss is data set aggregation. During the performance analysis and visualization process, the user must be able to view performance data at a wide range of time scales. Starting with an overview of the entire execution, the data exploration process may lead the user to a focused view of a few thousand cycles. In order to support the continuous feedback demanded by dynamic queries, a visualization system needs to quickly generate visual representations of the events occurring within the specified time window. Rivet achieves this functionality by computing an aggregation structure for each data object.

Figure 2(a) illustrates this aggregation structure. The raw data consists of discrete events, each with a beginning and ending cycle. The subsequent levels of the aggregation structure are built by dividing the execution time into discrete intervals. For each time interval, Rivet computes the fraction of time occupied by each different event type, as shown in Figure 2(b).

The choice of whether to use raw data or aggregated data, and the choice of what level of aggregation to use, are determined by three factors: the number of pixels available for display and the number of distinct events and cycles within the selected time interval. The system weighs these three factors to select a default set of data to be displayed. The raw data is displayed using Gantt charts, and the aggregated data is displayed using utilization strip charts. The user is able to override the selected level of aggregation if desired. Aggregation of events in the time dimension is useful for showing trends in the data when looking at the overall behavior of the application.

In addition to aggregation over time, we also perform aggregation across data objects, such as combining the data from a group of processes to produce a single data set for an entire application or subsystem. As we will show in the case study, accumulating the various Argus processes into three summary data sets enables us to see the behavior patterns of the three Argus process classes and guides us to regions of interest in the individual process views. This sort of aggregation will become increasingly important as we investigate the performance of larger multiprocessor systems, where there are so many elements that they cannot all be displayed individually.

4. Case study: Argus performance analysis

The necessity of the three features discussed above – flexibility, exploration and aggregation – can easily be seen in our performance analysis of the Argus parallel rendering library. In this section, we describe the development of the parallel performance analysis system, driven by the study of Argus. We begin by providing background on Argus itself. We then describe the simulations we performed to study the behavior of Argus, the visualizations we developed to analyze the data collected under simulation, and the discoveries and improvements we made during the analysis process.

4.1. Argus background

Argus [9] is a parallel immediate-mode rendering library targeted at large shared memory multiprocessors. An application developed with Argus consists of one or more relatively heavyweight processes. Argus operates entirely within a dedicated region of shared memory that it allocates itself. This allows for the possibility of user applications based on separate memory spaces (e.g. using UNIX *fork*) or on a single shared memory space (e.g. using UNIX *sproc*).

The driving application used here is a parallel NURBS patch renderer. The application creates multiple processes, each of which takes patches from a central work queue to simultaneously tessellate and render. The model used in this study has 102 patches, each consisting of 196 control points, and is tessellated into 81,600 triangles.

Argus is internally multithreaded, using a custom lightweight thread system. Argus does not have an internal notion of processors, but rather treats the individual user processes as though they were processors. When a user process first calls the library, Argus creates a lightweight thread to continue the user code, as well as a number of other threads to perform various rendering tasks. Argus controls the scheduling of these threads within each of the original user processes. Machine-dependent optimizations relating to placement and scheduling of processes onto processors are left to the application.

Argus creates threads for five types of tasks. The three primary tasks are: (a) the user's application work, (b) geometry processing, and (c) rasterization processing. An *app thread* is created to continue execution of the user's application code in each of the original user processes. Multiple *front threads* are created to perform geometry processing on graphics primitives. The framebuffer is subdivided into small tiles, and one *back thread* is created to handle rasterization of primitives into each tile. In addition to these three types of threads, one *submit thread* is created per app thread to control the parallel issue of graphics commands from all of the app threads according to user-specified constraints as described in Igehy et al. [9]. Finally, multiple *reclaim threads* are created to perform internal bookkeeping for shared data structures.

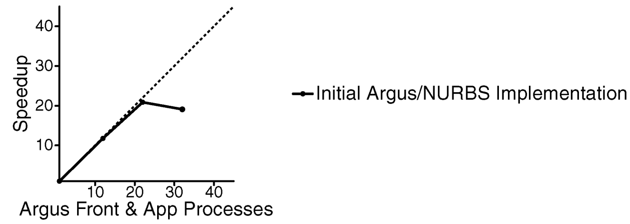


Figure 3: Initial speedup curve for Argus on SimOS.

Scheduling can be controlled by the user at the level of allowing or disallowing the different types of work on each process. For this study, we divide the processes into three categories: *back processes* are dedicated to rasterization, *front processes* are dedicated to geometry processing, and *app processes* may perform either application work or geometry processing.

The Argus configuration studied in this paper models a system with fast dedicated rasterization nodes rather than software rasterization. The primitive data and graphics state information normally needed for rasterization is referenced by the back threads to ensure its transfer to the rasterization node (as would occur in a hardware rasterization system), but software rasterization is otherwise disabled. Since the back threads are effectively acting as data sinks in this configuration, the goal is to achieve linear speedup in the number of front and app processes, ignoring the back processes.

4.2. Simulation environment

Argus was originally developed on an SGI Challenge multiprocessor, and without significant problems was able to achieve linear speedup running the NURBS application (implemented using *fork*) up to eight processors (the size of the machine used). To study the scalability further, the Argus developers ran the same application on an SGI Origin system with 64 processors. They immediately encountered a common performance problem for parallel applications running on this class of multiprocessor: longer and non-uniform memory latencies. These problems were resolved by adding prefetch instructions to Argus and by improving the memory reference locality. This enabled Argus to achieve linear speedup up to 31 processes (26 front & app processes, five back processes) [9]. Beyond that point, however, the performance diminished rapidly, and they soon observed slowdowns as they added processes. They utilized standard performance debugging tools such as software profiling and hardware performance counters, but were unable to discover the performance bottleneck.

In order to analyze this behavior in more detail, we configured SimOS to model a 40-processor Origin-type system running IRIX 6.4 (the SGI implementation of UNIX) and performed the same set of test runs under simulation. We observed the same overall performance behavior when running the application on SimOS as on the real Origin system. The speedup curve for the NURBS application running on SimOS is shown in Figure 3. We focused our attention on a 39-process Argus run (34 front & app, five back), a scale sufficiently large to exhibit slowdowns over runs with fewer processes.

4.3. Memory system analysis

We initially speculated that the memory system was the performance bottleneck. On systems like the SGI Origin, cache misses to remote nodes can become increasingly expensive as system sizes grow, and contention in the memory system can increase the cost of memory accesses. To investigate this hypothesis, we added a set of annotations to SimOS to collect detailed memory system statistics, and we developed a set of displays in Rivet to present this data.

MView: Memory System Visualization. The MView memory system visualization is presented in Figure 4. MView is composed of several views providing detailed information about the memory system behavior of the application.

In the *code view*, MView displays memory stall time organized by source code file and line. Simple bar charts are used to indicate the total percentage of memory stall that can be attributed to each source file. Below, misses are displayed by source code line using a display based on the *SeeSoft* [6] system. Lines of code that suffered cache misses are highlighted according to the amount of stall time incurred by the line.

The *memory view* shows memory stall time by physical and virtual addresses. Histograms are drawn for each physical node, with stall time classified into bins of 16 pages (256K). Another histogram is used to depict memory stall by virtual memory address, also with stall time binned into sets of 16 pages. Clicking on a histogram entry

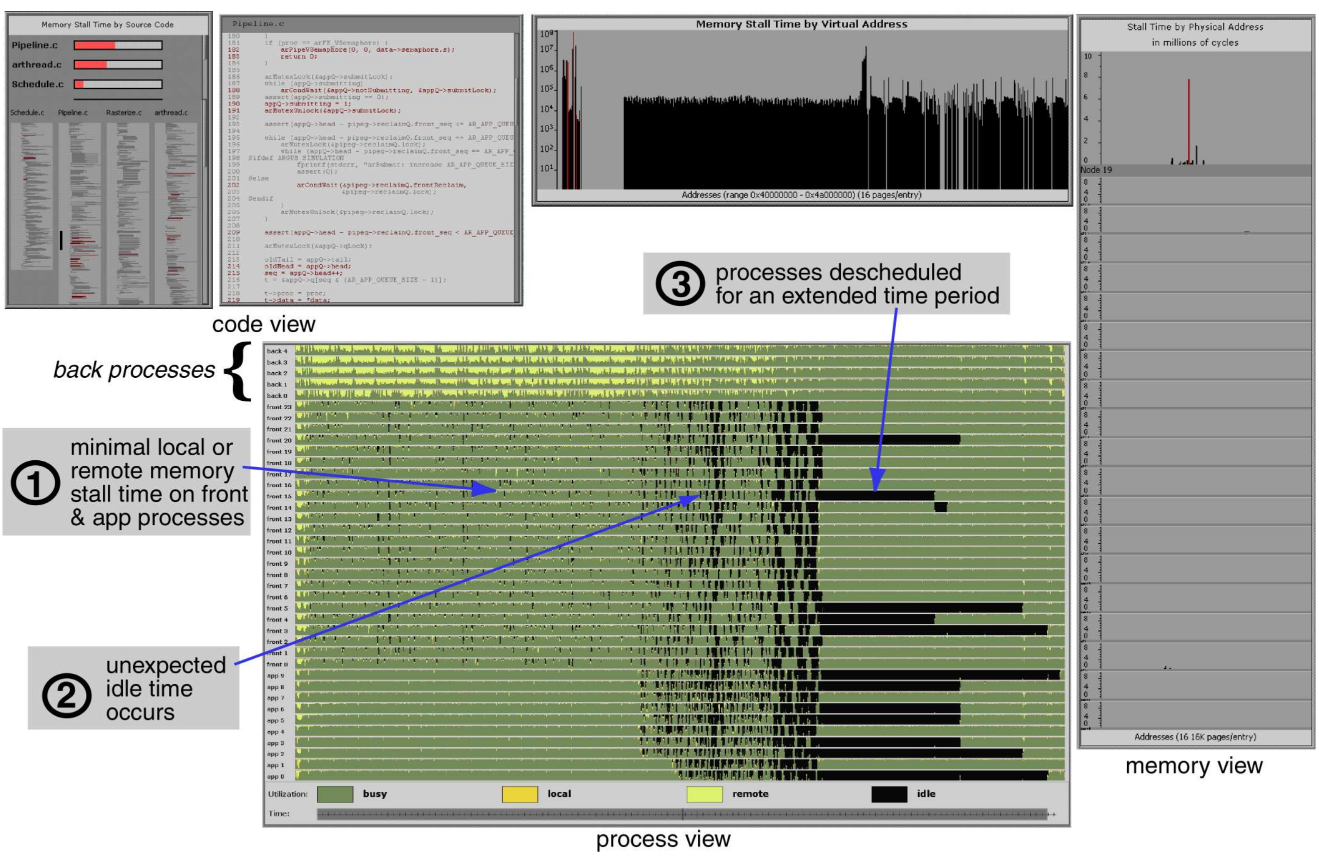


Figure 4: The MView visualization for a 39 process Argus run. This visualization depicts memory stall time by source code line (code view) and by physical and virtual memory address (memory view). It also shows process activity (busy or idle) and memory stall for each process as a strip chart in the process view. The memory view appears to show considerable stall time for the virtual addresses, but this is because this data is being displayed using a log scale.

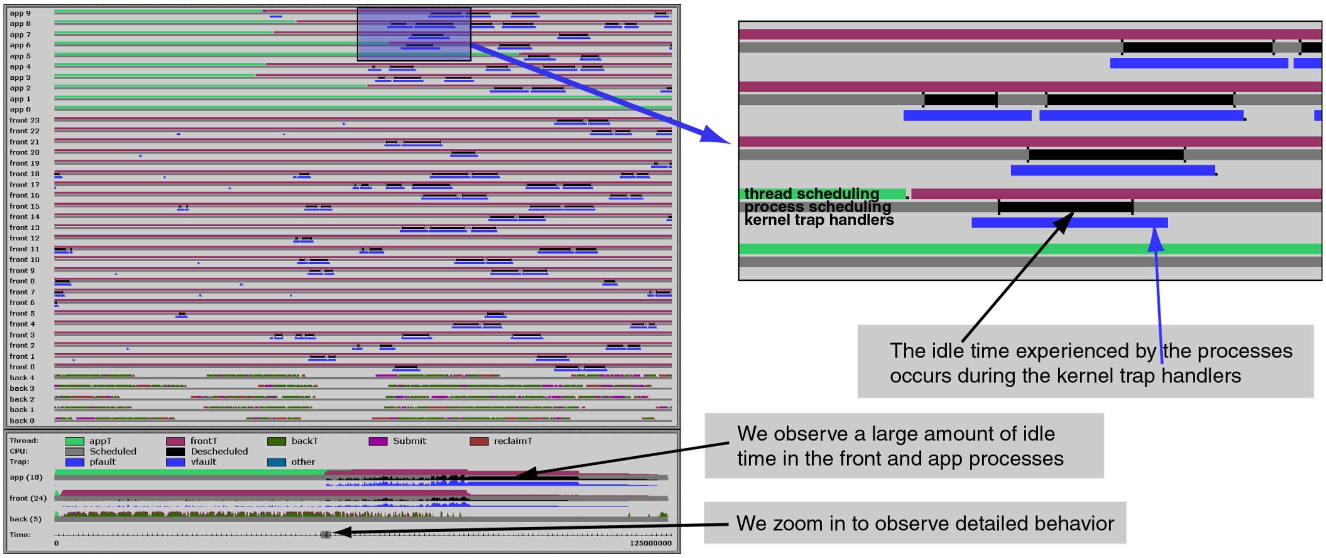


Figure 5: The PView visualization of Argus with 39 processes. The upper window shows data organized by process; the lower window presents the same data aggregated by Argus process class (app, front, back). Multiple charts can be displayed for each process – in this figure, we display thread scheduling, process scheduling and kernel trap information. The summary window displays all 124 million cycles of the run; the process view has been zoomed in to show 2 million cycles (10 milliseconds) of execution.

in either the physical or the virtual memory view will highlight the corresponding pages in the other view. The use of highlighting in this view is valuable for understanding placement problems, as the user can easily identify which pages are hotspots, the nodes on which those pages reside, and their corresponding virtual addresses. With additional annotations, these virtual addresses could be mapped to application data structures

Processor utilization for each Argus process over time is shown in the *process view*. A strip chart is drawn for each process, with time progressing from left to right. The chart indicates the fraction of time the process spent doing useful work, waiting for requests to the local and remote memories, and descheduled.

A legend and a time control appear below the process view. The legend identifies the color scheme in the process view and allows the user to make changes to the scheme. The time control affects both the process view and the memory view. This control allows the user to navigate from a macroscopic view of millions of cycles to a microscopic view of only a few thousand cycles. All of the charts can be rearranged, sorted and resized by the user.

Results of MView Analysis. The MView visualization of the Argus data quickly enabled us to realize that our initial hypothesis was wrong. Had memory stall been the cause of the performance problem, the strip charts shown in the process view would have contained significant sections of local and remote memory stall time. Instead, they showed almost full CPU utilization for the first two thirds of the run. In fact, the memory stall for the entire run accounted for only 3.9% of the total execution time of the app and front processes.

While MView enabled us to rule out the memory system as the cause of the performance problem, it also showed us a surprising pattern of behavior. Towards the end of the run, many processes began to experience significant amounts of idle time, culminating in several processes being descheduled for a long period. All told, the application spent 16.3% of its total execution time descheduled. This substantial amount of time spent descheduled would account for the poor performance of Argus at this scale.

This observation was totally unexpected. The Argus internal scheduler never yields the CPU voluntarily – an Argus process will go idle only if the kernel explicitly deschedules it. Since there were no other active processes in the system, there was no obvious reason that the kernel should have prevented the Argus processes from running.

4.4. Process Scheduling Analysis

To understand the kernel's process scheduling behavior, we repeated the Argus run, focusing our data collection on the processes' interactions with the operating system. We added several annotations to SimOS to collect information on process scheduling, kernel exception handling, and Argus thread scheduling. To display this data we developed a new visualization, an evolution of the MView process view.

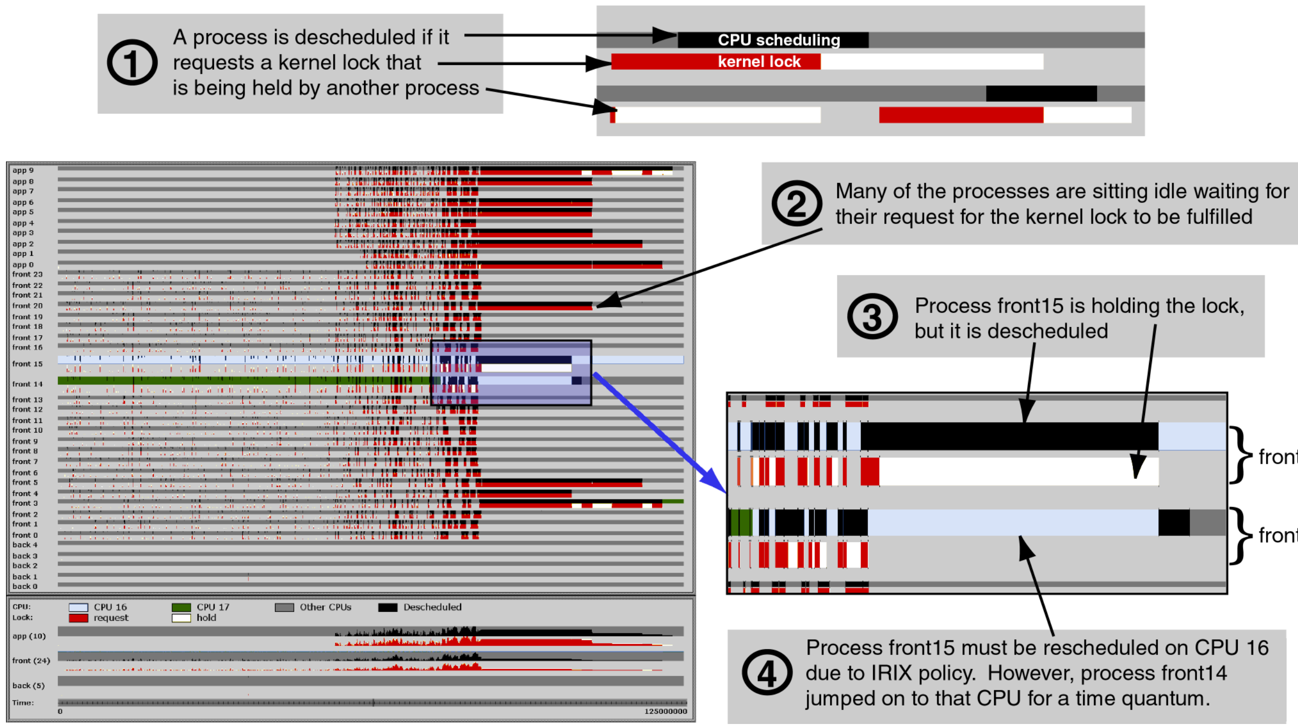


Figure 6: PView visualization of a 39 process Argus run showing CPU scheduling and kernel lock information. About halfway through the run there is a substantial increase in lock contention. During this period, one of the processes (front15) is granted the lock while descheduled. The CPU that the process must be scheduled on is being used by another process (front14). Therefore, process front15 holds the lock for an entire time quantum; meanwhile, many of the other processes must be descheduled until the lock is available.

PView: Process Visualization. The PView process visualization is shown in Figure 5. This visualization was designed to allow us to study a variety of per-process attributes varying over time.

The top window in Figure 5 displays the data organized by process. For each process, three charts are displayed. All the charts share a common time axis, running from left to right. We decided that the different data sets for a single process should be placed together. This is in contrast to the MView visualization, which allocated each type of data to a separate view. By using visually distinct color sets for different process attributes, we found it possible to make visual comparisons of the different attributes of a particular process (by adjacency) and of a particular attribute across all processes (by focusing on a particular color set).

The raw data displayed in these charts consists of sets of discrete events, each with a beginning and an ending cycle. We use the data aggregation mechanism described in Section 3.3 to display this data as either a Gantt chart or a utilization strip chart, depending upon the number of cycles being displayed.

The bottom window provides an overview of the per-process data. The same information is shown, but aggregated into three sets of processes corresponding to the three process classes in Argus (app, front, back). This overview allows the user to quickly identify characteristics specific to a single type of process and to gain a summary of the entire execution. This window also contains the same time control as the MView visualization. The time control affects only the per-process view; the overview always shows the entire data set, providing context and serving as a guide for the time control.

Results of PView Kernel Analysis. Figure 5 shows the PView display with three sets of data – thread scheduling, kernel traps, and process scheduling. Examining the trap data alongside the process scheduling data, we saw that the idle time occurred during the kernel *pfault* and *vfault* routines. These kernel routines are used for updating process page tables; in this case, the processes were faulting on pages in the shared memory region used by Argus for common data and synchronization. By examining the thread scheduling information in the summary window, we saw that the idle time became significant when those processes running *app* threads switched to running *front* threads. Having all of the data for a process in a single view, along a common time axis, allowed us to uncover several relationships that would have been difficult with separate views.

During the early portions of the run, these kernel traps occurred relatively infrequently and with short duration. In

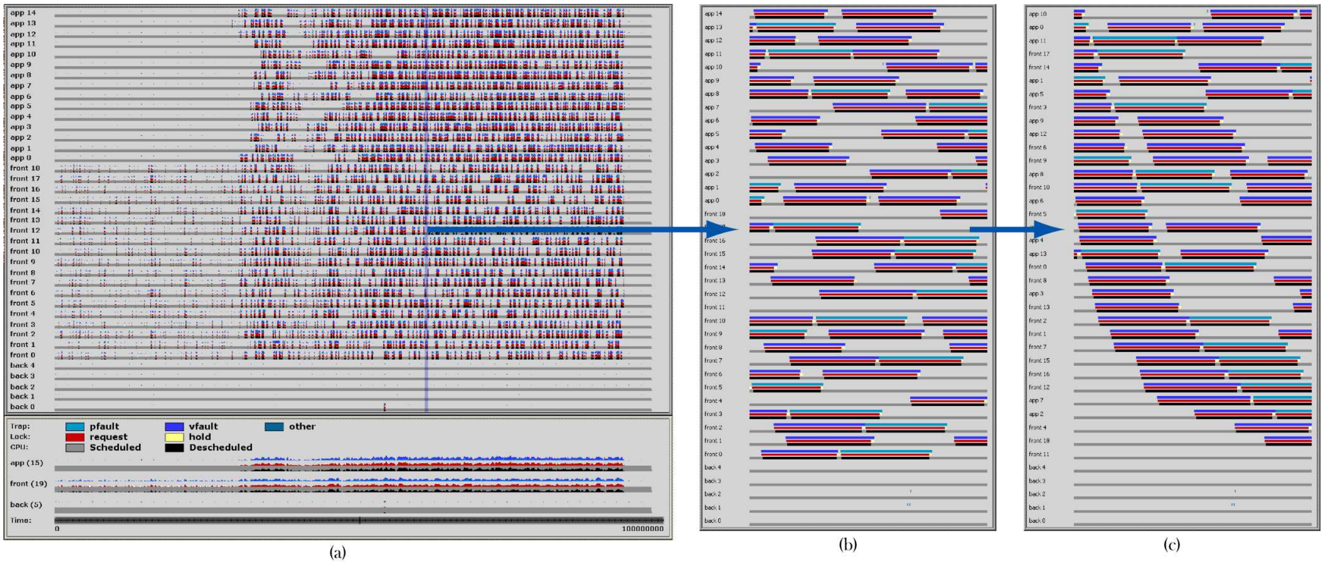


Figure 7: Three successive screenshots of the PView visualization of a 39 process Argus run with processes pinned to prevent migration, showing kernel trap, kernel lock and process scheduling information. While pinning the processes prevented the process scheduling quirk, we still see substantial trap activity, lock contention, and idle time in (a). The display is zoomed to a small time window in (b), which shows that both the idle time and the kernel traps correspond to the time spent requesting and then holding the lock. The time spent requesting the lock far exceeds the time spent holding it because of the contention. Figure (c) shows the same time window as (b) after the user has interactively sorted the charts to see the progression of the lock from process to process.

the latter portions of the run, they appeared more frequently and with longer duration. Examination of the IRIX source code showed that faults on shared memory regions require locking, leading us to suspect that kernel synchronization was the cause of the idle time. To investigate this possibility, we performed a third Argus simulation, this time adding annotations to collect information on the kernel's shared memory lock. The results of this simulation are shown in Figure 6. In the figure, PView displays the CPU scheduling data along with kernel lock data indicating times the process was either waiting for or holding the lock. By examining the lock data next to the scheduling data, we were immediately able to confirm our hypothesis: lock contention was the reason for the descheduling of the Argus processes. A process would request the lock, find it was already held by another process, and be descheduled until it was granted the lock by the kernel.

In general, the lock was only held for a short period before being released. In one case, however, we saw the lock held for an extended period. This explained the long stretches of idle time we observed in the original visualization: several of the processes were suffering faults, queuing on the shared memory kernel lock, and being descheduled until they could acquire the lock. However, it did not explain why one process was holding the lock for such a long time in this one instance.

By comparing the lock and CPU scheduling charts for the process in question, we observed that it remained descheduled for an extended period even after it was granted the lock. To understand this behavior, in the process scheduling view we highlighted the CPU on which the process was initially scheduled. We observed that, while the process was waiting to acquire the lock, another process was scheduled on its processor. This should not have been a problem, since there were other processors available to run the process once it received the lock.

However, further examination of the kernel fault routines showed that the kernel *pfault* and *vfault* routines must pin the faulting process to its CPU in order to prevent process migration during trap handling. Consequently, the process could not be scheduled anywhere else and was forced to wait for the other process to be descheduled. Since Argus processes do not voluntarily yield the CPU, it ran for an entire time quantum (20 million cycles) before being descheduled by the kernel. At that point, the original process was scheduled, quickly finished handling the fault, and released the lock.

This scheduling behavior can be avoided by explicitly pinning the Argus processes to specific CPUs. We ran Argus on SimOS again, this time using SGI's *dplace* tool to prevent process migration; the results are shown in Figure 7. The figure shows that pinning the processes eliminated the large section of idle time caused by the process scheduling quirk. However, there was still a significant amount of idle time during the latter stages of the run caused by contention on the same kernel lock. Figure 7 shows how we used the interactive layout support provided by Rivet to observe this behavior. We zoomed in on a very small section of the execution to see the correlation between process idle time and individual kernel lock re-

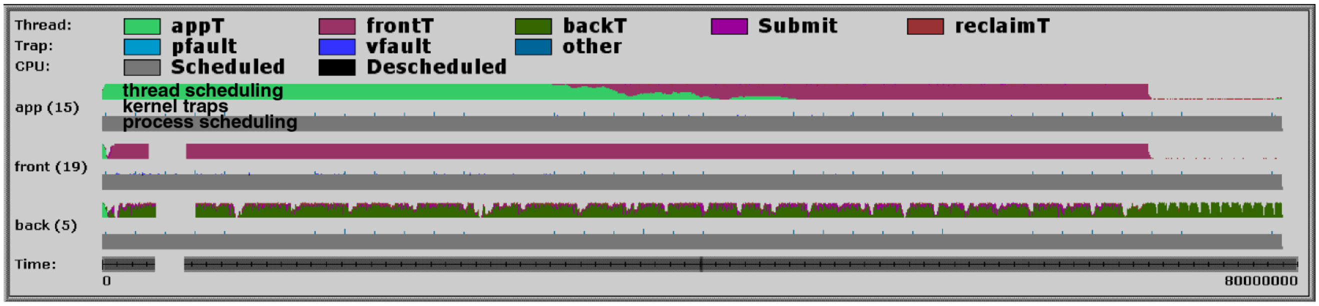


Figure 8: PView summary view of thread scheduling, trap handling, and CPU scheduling information for a 39 process Argus run after the completion of the performance analysis. There is little time spent descheduled or handling kernel traps. This run completes nearly twice as fast as the initial Argus run, achieving 95% of linear speedup.

quests. We then rearranged the process layout by direct manipulation to see the heavily contended kernel lock being passed from one process to another in a first-come first-served fashion.

4.5. Changing the multiprocessing mechanism

As noted above, the performance bottleneck that was limiting the scalability of Argus was a lock used by the kernel to control access to the internal data structures of the shared memory region. The use of this shared region was necessitated by the decision to implement the NURBS application as a set of independent processes with distinct address spaces using the *fork* multiprocessing mechanism. While this was not a problem when running smaller numbers of processes, contention for this resource quickly became the limiting factor at larger process counts.

Having learned that this was the bottleneck, we sought an alternative that would support shared memory without the coarse kernel synchronization of a single shared memory region. We considered manually dividing the shared region into several smaller subregions. However, this would not necessarily be effective if concurrent accesses occurred to the same subregion, and it would have added significant complexity to the implementation of Argus.

Instead of subdividing the shared region, we decided to eliminate it altogether: we modified the application to use the *sproc* multiprocessing mechanism instead of *fork*. This mechanism, which is optimized for multithreaded systems like Argus, creates a set of processes which share a common address space. This allows the Argus processes to communicate and share data without the explicit use of a shared memory region, providing synchronization at a much finer grain within the kernel.

We ran the new version of the NURBS application on SimOS using the same configuration as the preceding runs; the PView summary window is shown in Figure 8. In this run, the amount of kernel trap handling was reduced to a negligible level, and all processes in the system remained scheduled and busy for the duration of the run. As shown in Figure 9, this run completed nearly twice as fast as the initial Argus run and achieved 95% of linear speedup. Furthermore, with this version of the application we were able to achieve 90% of linear speedup all the way up to 45 front and app processes (with 11 back processes).

4.6. Summary

As the performance analysis of Argus has demonstrated, the coupling of simulation with visualization can generate an extremely powerful performance analysis tool. We were able to uncover subtle interactions between the graphics library and operating system that would have been extremely difficult to discover using traditional tools.

During the first phase of the Argus study, we performed three distinct simulation runs of the NURBS application, with each run using the exact same application and configuration but a different set of annotations. The determinism of SimOS ensured that each run produced identical results, and the flexibility of Rivet enabled us to easily incorporate the new data into our visualizations.

After several sessions of simulation and visualization, we achieved insight about the performance bottlenecks that were limiting the application's performance. We used that knowledge to refine the implementation and configuration of the program and ran the application once more to confirm that that particular performance problem had been solved. At that point, however, the process started all over again with the next performance bottleneck.

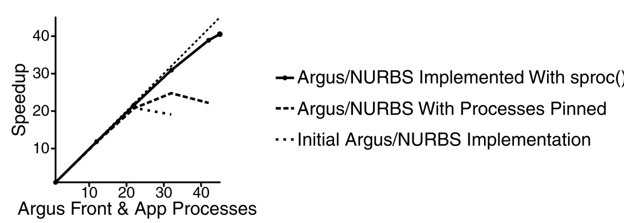


Figure 9: Speedup curves for the three versions of the Argus NURBS application, showing the performance and scalability improvements achieved by changes made during the analysis and visualization process.

5. Discussion

Coupling Rivet with SimOS has proven to be a powerful technique for performing parallel systems performance analysis. However, two potential limitations are simulation speed and fidelity, and the small number of reusable visualization scripts and annotations written thus far.

Because detailed simulation is significantly slower than regular program execution, simulation is most applicable to the study of several seconds or minutes of execution, not several hours or days. This problem can often be overcome by using faster, less detailed simulation modes to reach the section of interest [8].

Another concern with simulation is that most simulators do not fully model the detailed behavior of real hardware. Typically, if a problem is observed in the simulator it also occurs on the hardware, but the converse is not necessarily true. In the case of Argus, after fixing the problems found in the case study, the pathological lock behavior was no longer observed on the real hardware; however, the application did not scale as well on the hardware as it did in the simulator. This is likely due to memory system issues: since SimOS uses a generic NUMA model and not a detailed model of the Origin, it is possible that subtle memory effects are not being modeled in SimOS. In this situation, Rivet can still be used by importing data from another source such as hardware counters.

A final limitation is the lack of pre-built annotations and visualizations. While the flexibility of Rivet and SimOS enabled us to adapt our data collection and visualization scripts as we uncovered problems in Argus, many parallel systems problems recur across applications. The existence of a configurable annotation and visualization library would simplify the task of analyzing these common problems. We expect that as Rivet and SimOS are used in further studies, focused scripts like the ones used in this case study will be generalized into such a library.

6. Conclusion and future work

We are developing a system for the analysis and visualization of parallel systems. This framework, which couples a complete machine simulator with a flexible visualization environment, provides support for iterative performance analysis. In this paper, we have described how this system has been used for the study of Argus, a parallel graphics rendering library. At the completion of this study, Argus was able to achieve 90% of linear speedup with up to 45 processes. In addition to improving the performance of Argus, this performance analysis has helped drive the development of our system.

The system demonstrates how visualization can be used to realize the full power of simulation, creating a powerful and effective tool for performance analysis. The design of our system also illustrates several issues important to the integration of visualization and simulation. The visualization system must be extremely flexible to display the wide variety of information that can be collected. It must also provide support for interactive layout and dynamic queries, enabling the user to effectively explore and search the large amount of data that can be generated during simulation of a parallel application. Finally, data aggregation schemes must be used to enable the visualization system to support both efficient navigation and cycle-accurate display for data sets containing millions of cycles of data. We demonstrate one such aggregation scheme.

Future work on the analysis and visualization framework will be focused on the Rivet visualization environment. The compositional architecture of this environment provides a degree of flexibility that has proven very valuable in our study of computer systems. We will continue to develop this architecture, exploring innovative composition techniques for both visual and data primitives. The aggregation mechanisms utilized in this study have also proven extremely effective. We intend to extend these aggregation techniques for use on a wider range of data types, and to further explore how aggregation can enable data displays to adapt to available resources such as display technology and processing power.

Acknowledgments

The authors thank Kinshuk Govil and David Ofelt for their help with the case study. We also thank Diane Tang and Tamara Munzner for their work reviewing this manuscript. Finally, the contributions of the anonymous reviewers and our shepherd, Hans Eberle, were invaluable.

References

[1] L. Barroso, K. Gharachorloo, and E. Bugnion. "Memory System Characterization of Commercial Workloads." *Proc. of the 25th International Symposium on Computer Architecture*, June 1998.

[2] J. Bertin. *Graphics and Graphic Information Processing*. Berlin: Walter de Gruyter & Co., 1981.

[3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors." *ACM Transactions on Computer Systems*, 15:4, November 1997.

[4] W. Clark. *The Gantt Chart*. London: Sir Isaac Pitman & Sons Ltd., 1952.

[5] L. DeRose, Y. Zhang, and D. Reed. "SvPablo: A Multi-Language Performance Analysis System". *Tenth International Conference on Computer Performance Evaluation - Modelling Techniques and Tools – Performance,* September 1998.

[6] S. Eick, J. Steffen, and E. Sumner, Jr. "SeeSoft – A Tool for Visualizing Line-Oriented Software Statistics." *IEEE Transactions on Software Engineering,* 18(11):957-968, November 1992.

[7] M. Heath and J. Etheridge. "Visualizing the Performance of Parallel Programs." *IEEE Software*, 8(5):29-39, September 1991.

[8] S. Herrod. "Using Complete Machine Simulation to Understand Computer System Behavior." Ph.D. Thesis, Stanford University, February 1998.

[9] H. Igehy, G. Stoll, and P. Hanrahan. "The Design of a Parallel Graphics Interface." *Proc. SIGGRAPH '98*, pp. 141-150, August 1998.

[10] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server." In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pp. 241-251, May 1997.

[11] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. **"**The Paradyn Parallel Performance Measurement Tools.**"** *IEEE Computer* 28(11):37-46, November 1995.

[12] W. Nagel and A. Arnold. "Performance Visualization of Parallel Programs – The PARvis Environment." *Proc. of the 1994 Intel Supercomputing Users Group Conference*, pp. 24-31, 1994.

[13] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L. Tavera. "Scalable Performance Analysis: The Pablo Performance Analysis Environment." *Proc. of the Scalable Parallel Libraries Conference*, pp. 104-113, 1993.

[14] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, and A. Gupta. "The Impact of Architectural Trends on Operating System Performance." *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[15] B. Shneiderman. "Dynamic Queries for Visual Information Seeking." *IEEE Software*, 11(6):70-77, 1994.

APPENDIX: SimOS and Rivet scripts – An example

This is a simple example of the scripts used to collect and display data in SimOS and Rivet, consisting of a set of SimOS annotations that collect Argus thread scheduling data and a Rivet script to display this data as a collection of Gantt charts.

SimOS Annotations

```
annotation set pc Argus::arScheduler:START {
  log "<THREAD> deschedule: "
  log "$CPU $PID($CPU)-$PROCESS($CPU) $CYCLES\n"
}
```

Process enters the Argus thread scheduling routine (arScheduler): Report that the currently active thread is being descheduled.

SimOS exports hardware state through Tcl variables:
Processor number (`CPU`), cycle count (`CYCLES`), CPU registers (`v0`)
Kernel annotations included with SimOS track kernel state:
Currently scheduled process (`PROCESS($CPU)`) and PID (`PID($CPU)`)
Annotations may refer to application addresses using symbolic names:
Instruction (`arScheduler:START`) and data (`((arThread*)$v0)->name`)

```
annotation set pc Argus::arScheduler:END {
  set thread [symbol read Argus::((arThread*)$v0)->name]
  log "<THREAD> schedule: "
  log "$CPU $PID($CPU)-$PROCESS($CPU) $CYCLES $thread\n"
}
```

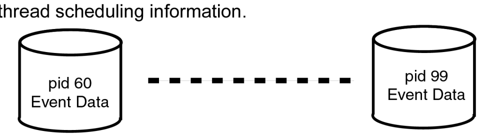
Process exits the Argus thread scheduling procedure (arScheduler): Report the scheduling of the newly activated thread.

```
...
<THREAD> deschedule: 16 84-NURBS 9065184
<THREAD> schedule: 9 91-NURBS 9067204 back(pipe:1,tile:9)
<THREAD> schedule: 17 76-NURBS 9094305 reclaim(pipe:1)
...
```

Rivet Visualization Script

```
for {set pid 60} {$pid < 100} {incr pid} {
  set argus_threadData($pid) [EventData]
}
```

Create a data object (EventData) for each process to store the thread scheduling information.




```
set parser [Parser]

$parser AddString                                        \
  THREAD                                                 \
  {^ schedule: [0-9]+ ([0-9]+)-NURBS ([0-9]+) (.+)$}     \
  Schedule
rbind $parser <Schedule> ArgusThreadSchedule

$parser AddString                                        \
  THREAD                                                 \
  {^ deschedule: [0-9]+ ([0-9]+)-NURBS ([0-9]+)$}        \
  Deschedule
rbind $parser <Deschedule> ArgusThreadDeschedule
```

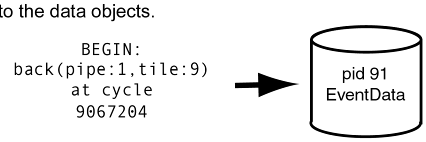
Use regular expressions to describe the log file to the parser. The regular expression extracts the process id, cycle and thread name and passes them to the ArgusThreadSchedule and ArgusThreadDeschedule procedures.

```
$parser AddString
  THREAD
  {^ schedule: [0-9]+ ([0-9]+)-NURBS ([0-9]+) (.+)$}
  Schedule

<THREAD> schedule: 9 91-NURBS 9067204 back(pipe:1,tile:9)
```

```
proc ArgusThreadSchedule {pid cycle thread} {
  global argus_threadData
  $argus_threadData($pid) BeginEntry $thread $cycle
}
proc ArgusThreadDeschedule {pid cycle} {
  global argus_threadData
  $argus_threadData($pid) EndEntry $cycle
}
```

Define the procedures that the Parser calls when matches are found in the log file. `BeginEntry` and `EndEntry` add data to the data objects.




```
rparse $parser "/u/bosch/argus/sproc.log"
```

Request that the parser perform the actual parsing of the file.

```
rglob GLWindow .threads
.threads SetLayoutMgr [StackLayoutMgr]
```

Create a window that will vertically stack all objects placed into it.

```
foreach pid [array names argus_threadData] {
  rglob Gantt .threads.$pid
  .threads.$pid SetData $argus_threadData($pid)
}
```

For each process, create a Gantt chart in the window and set its data source to be the corresponding event data that we just created.