PEACOCK — A *Programmer's Cockpit* for Comprehensive Program Comprehension

Christian S. Collberg John H. Hartman Stephen Kobourov

A Cover Page

B Project Summary

The goal of this project is to use inexpensive but massive computational resources and displays to enhance large program comprehension.

B.1 Intellectual Merit

Real world programs are large, complex, evolving, distributed, interactive, and resource-demanding. They are built by continuously evolving software engineering teams with members of varying skill levels, and they are kept up-to-date by different teams of software maintenance engineers. This proposal is concerned with gathering information — static, dynamic, and historical — about real world software systems and presenting it in a simple and coherent way to various types of engineers during development, debugging, and maintenance phases. The vision is a system — a *Programmer's Cockpit* — that integrates what is traditionally thought of as disparate activities: browsing program text, debugging, and performance evaluation and tuning. The analogy is with a traditional airplane cockpit in which numerous dials present a comprehensive view of the health and status of the plane. In particular, the Programmer's Cockpit will gather performance data, extract historical information from a version control database, visualize the program text and structure, visualize performance data, and allow engineers to navigate through program execution, using one integrated tool.

Previous work in program comprehension has sacrificed interactivity, accuracy, and presentation to run on stock hardware. In contrast, this proposal will examine the use of inexpensive but massive computational resources and displays, allowing large, highly interactive programs to be examined, while collecting accurate performance data at all levels. A novel geographic software visualization metaphor (based on computer game design and implemented using publicly available game engines) will be examined, allowing the huge amounts of data (static, dynamic, and historical) that will be gathered to be effectively visualized.

B.2 Broader Impact

The virtual worlds created by the computer game industry have become powerful metaphors well understood by a large segment of the population. Not only young adults but people in all walks of life are skilled in the use of game controllers and joysticks to maneuver through complex three-dimensional virtual spaces. Game designers, in turn, have become adept at the design of user interfaces that facilitate such maneuvering, for example allowing users to rapidly shift between local, global, first-person, and third-person views of the terrain to prevent disorientation. We believe virtual worlds will become a powerful metaphor for visualizing data within a variety of fields, since they provide a rich array of objects that can be manipulated in an almost infinite number of ways. The tools, techniques, and experiences that come out of this research will be made available to the wider research community. They will be particularly useful as building-blocks in areas that need to visualize large amounts of data in real-time.

As an integral part of the project, we will involve graduate and undergraduate students in all aspects of the described research activities. This will continue a tradition, in our research groups, of integrating research activities into the undergraduate curriculum; involving undergraduates into our research; developing research-based educational materials in order to incorporate research into learning and education; and formulating and disseminating innovative and effective approaches to science teaching. We anticipate that our use of a virtual worlds metaphor will attract undergraduate students to join in our research effort. We will particularly target minority students who might not otherwise have been interested in or exposed to research projects.

We will also make software developed as part of the project available to the broader research community. This will, in particular, benefit institutions and individuals lacking the infrastructure or resources to develop such software tools for themselves.

C Table of Contents

D Project Description

D.1 Introduction

Real world programs are large, complex, evolving, distributed, interactive, and resource-demanding. They are built by continuously evolving software engineering teams with members of varying skill levels, and they are kept up-to-date by different teams of software maintenance engineers. During the lifetime of the program different questions are asked about it: “How did the program evolve into its current module structure?” “What is the source of this performance problem?” “How can I find the source of this bug in millions of lines of unfamiliar code now that the original engineers have left the project?”

This proposal is concerned with gathering information — static, dynamic, and historical — about real world software systems and presenting it in a simple and coherent way to engineers during development, debugging, and maintenance phases. Our vision is a system that integrates what is traditionally thought of as disparate activities: browsing program text, debugging, and performance evaluation and tuning.

We will refer to the concept of comprehensive program comprehension as the *Programmer’s Cockpit*, and to our system as “PEACOCK.” The analogy is with a traditional airplane cockpit in which many dials present a comprehensive view of the health and status of the plane. Our system will similarly present the current state and performance characteristics of the program under study. It will also allow the user to browse the source code and examine the overall structure and relationships between different parts of the code, as well as viewing the historical development of the program. In particular, our system will consist of subsystems for gathering performance data, extracting historical information from a version control database, visualizing the program text and structure, visualizing the performance data, and navigating through the program execution. Figure 1 presents a high-level view of PEACOCK.

Our novel visualization framework will allow engineers to experience every aspect of a program using one simple metaphor. This includes visualizing the development history of the program (its version control database), its current static structure (the packages, modules, methods, and their interconnects), as well as its dynamic (performance) behavior. This is in contrast to previous work, in which different aspects of a program are visualized in different and *ad hoc* ways. We expect our system to be used by (a) engineers newly transfered to a project, to learn about its static structure; (b) maintenance engineers who need to track down bugs in unfamiliar code; (c) reverse engineers needing to gain a quick understanding of legacy code, including an understanding of its development history; and (d) performance tuning engineers who need an accurate view of the performance characteristics and bottlenecks of a program. In all cases they will need to learn only one tool and be presented with only one metaphor of the program and its behavior. In particular, we will be investigating the use of publicly available *computer game engines* to visualize every aspect of a program using a *geographic* metaphor.

Collecting run-time information of a program requires tremendous resources. Profiling and tracing information can be generated from multiple sources, including the CPU (performance monitors), an instrumented operating system, an instrumented virtual machine (such as the Java virtual machine, JVM), an instrumented program (Java bytecode, TCL scripts, binary executables), network packet sniffers, disk performance monitors, etc. (Figure 1 ⓑ). To get an accurate view of a system this data must be collected, collated, analyzed, and presented to the user. To prevent system overload, previous systems have typically refrained from collecting some data (potentially reducing the accuracy of the analysis), or resorted to a post-mortem analysis which does not work well for highly interactive programs. The goal of this project is to devise algorithms which allow the collection of every type of performance data, at every level of a computing system, without sacrificing accuracy or interactivity.

Previous work in this area has assumed a limited amount of computational power being available to do the data gathering and visualization. We consider this to be a case of misguided frugality. As an example, in Section G we show that for $35,000 it is today possible to purchase an 8-processor (2.2 GHz) computer with 16GB of memory, and a visualization system consisting of a high-end dual-processor workstation and

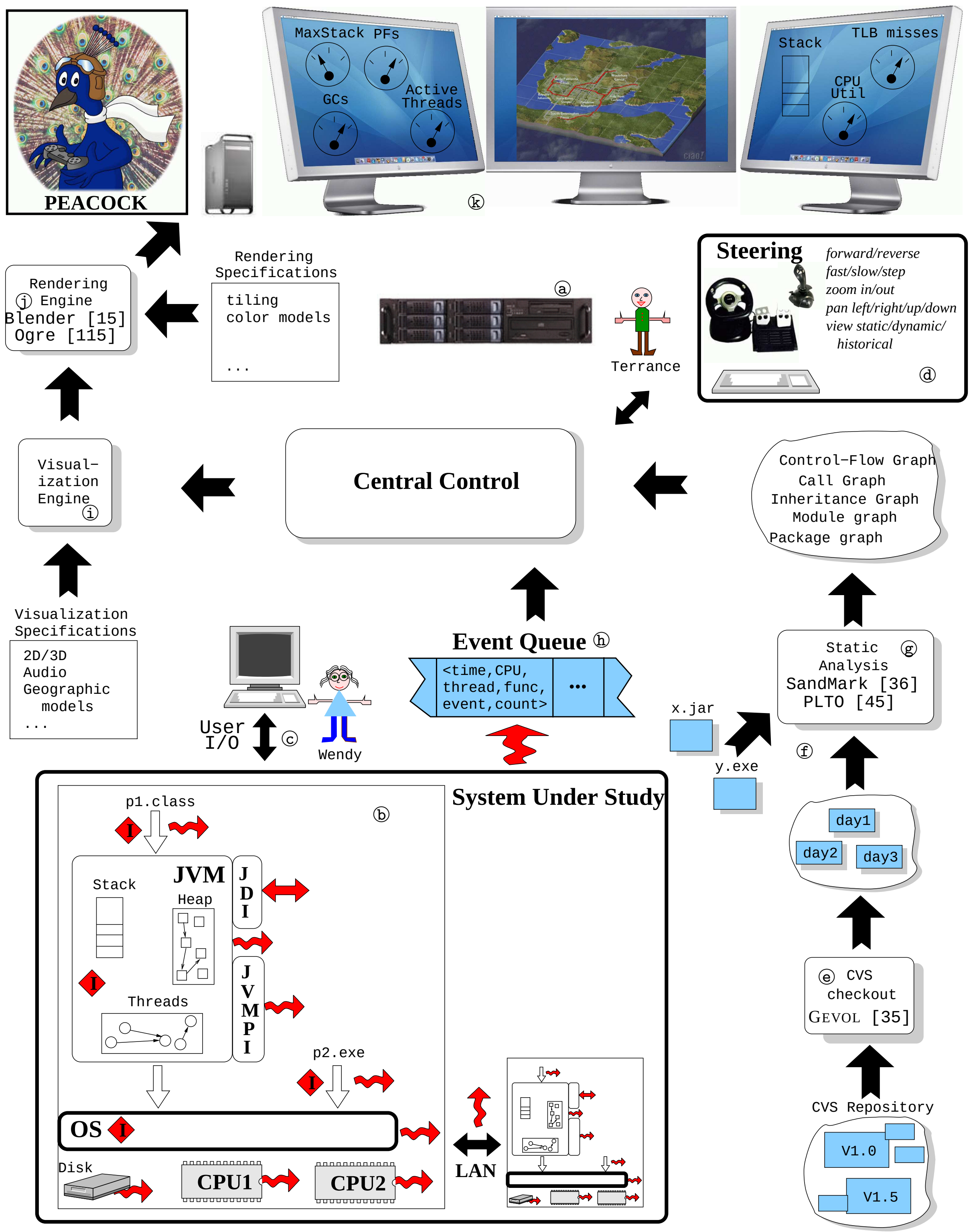

Figure 1: Overview of the proposed PEACOCK system. ◆ indicates an entity that can be instrumented to generate events. ↝ indicates an entity that generates events.

three 30-inch flat-panel displays. Amortized over three years, this is less than 12% of the salary of a senior software engineer in San Diego.[1] If a system such as proposed here can make the engineer even slightly more productive, it is well worth the investment. We will therefore make the assumption that massive computational resources and display real-estate will be available, and we will investigate how this can be best put to use to improve the comprehension of large programs.

To summarize, there are four main aspects of this proposal that set it apart from previous work:

1. Our system will collect performance data at all levels, without sacrificing accuracy.
2. Our system will target interactive programs, allowing performance data to be gathered without sacrificing the interactive experience.
3. We will use a single metaphor to present the massive amounts of data the system collects about a program, including performance data, historical development data, and data about the static structure of the program.
4. With the goal of improving program comprehension we will use massive amounts of inexpensive computational horsepower to accomplish the data gathering, analysis, interactivity and visualization, without compromising any of them.

D.2 Proposed Research

D.2.1 Overview

Much research has been done on tools for *software visualization*, *program comprehension*, *reverse engineering*, *performance profiling*, and *debugging*. Although targeting different aspects of the program life-cycle, these tools all have in common that they seek to discover information about a program. Unfortunately, the research communities that study the design of such tools are largely disjoint, and this has prevented a unified view of tools for program understanding. The goal of the research proposed here is to study the design of a tool that provides such a holistic view, helping a programmer to gain an understanding of the history, logic, and performance characteristics of a large software system. In addition to common tasks such as debugging and profiling, we expect our tool to be used in a purely *exploratory* way, allowing programmers to gain better intuition for the static structure and run-time characteristics of a program.

Let us begin by exploring a few scenarios in which the Programmer's Cockpit is helpful.

Scenario 1. Wendy has just been hired as a maintenance engineer. Her first task is to familiarize herself with the code base. She fires up PEACOCK, and takes a fly-through tour of the code. She starts by visualizing the code at a high level. As she flies through the code, she sees packages and classes laid out as a landscape of continents, countries and cities, with roads and rivers simulating interconnects. The recursive structure of the code (packages within packages, etc.) is naturally reflected in the recursive structure of geography (states within countrie, etc.). Close ties between pieces of code are reflected in their geographic proximity. She zooms in on a particularly large cluster of cities, noticing the highly complex network of roads that connect them. She makes a mental note to herself that this segment of the code may be a prime target for future refactoring. She continues zooming in on one of the cities (classes), examining each city block (method) in turn.

Scenario 2. Terrance, a programmer on a security-sensitive project, disappears under mysterious circumstances. The FBI fears the project may have been compromised, perhaps by the insertion of a Trojan Horse. The project's CVS repository is viewed in PEACOCK, highlighting all of Terrance's code changes. Did he add a piece of code (a possible Trojan) to a part of the system no longer under active development, to avoid scrutiny by other developers? The code is again visualized using a geographic metaphor: as

[1]In practice we expect a PEACOCK system to be shared between several engineers, making the cost essentially insignifi cant.

code grows and packages and classes split and merge over time, the corresponding continents and countries undergo continental drift and border changes. Next, the program is executed under PEACOCKfor various unusual inputs. A fly-through of the program "terrain" is viewed during execution, highlighting any code that Terrance wrote, but which is never executed on normal inputs — signs of a possible Trojan.

Scenario 3. Philip, a performance-tuning engineer for a computer game company, has received reports of a mysterious performance problem when gamers transfer from level 23 to level 24. He runs the game under PEACOCK, executing at near full speed. After an hour of play Philip crosses over to level 24, but notices nothing out of the ordinary. He uses PEACOCK's backwards execution facility to go back one minute in time, and continues forward again, this time picking a different way to kill Monster #354. This time, he does notice a remarkable delay in moving to level 24. The geographic visualization of the executing code uses light and sound to indicate activity level: cities light up as the corresponding code is executed, and the overall workload is represented by an audible humming sound. As expected, the city (class) representing level 24 is lit up brightly. Again, he goes back in time 10 seconds, and continues executing forward. This time PEACOCK catches all events and records all possible reasons for the delay. Philip notices that PEACOCK's performance dials (Figure 1 ⓚ) show a large increase in I/O around the time for the performance hit. He goes back in time once more, stepping forward through the program at a more detailed level. He comes to the routine `level_24_init`, and notices that the code that reads the level 24 scenery from disk under certain conditions will be called multiple times, rather than once, as expected.

Any specific program analysis task in the scenarios above may require several traditional tools, such as debuggers and profilers, to pinpoint a particular problem. The goal of the proposed research is to construct a system that allows an engineer to seamlessly move from task to task without having to move from tool to tool. However, there are many technical challenges that need to be addressed:

- Modern programs (games, simulators, browsers, instant messaging clients) are highly interactive. To do accurate performance measurements, the programmer must be allowed to interact with the program at near real-time speeds. At the same time, performance data needs to be captured at every level, from the CPU and disk to the operating system to virtual machines, and analyzed and visualized for the programmer. This requires high performance of PEACOCK itself.
- Modern programs are also long running, and performance problems (such as can be the results of slow memory leaks) may only manifest themselves after long periods of time. Once a problem is encountered, it is impractical to rerun the program from scratch. Rather, it becomes imperative to be able to execute backwards, and replay the offending part of the code. Reverse execution is complicated by the need to accurately reproduce all internal and external events, including operating system scheduling decisions, network traffic, and disk I/O.
- Tremendous amounts of execution and performance data will be gathered during long program runs and presenting this to the programmer in an easily digestible format will be challenging.
- When the system under study is highly interactive (such as immersive simulations or computer games) it will not be possible for one person to simultaneously interact with it and PEACOCK. It will be necessary to study the interaction between two or more users (Wendy and Terrance in Figure 1), driving the system under study and driving and monitoring PEACOCK itself.

Figure 1 gives a high-level overview of PEACOCK. Major components of the system are a high-end multiprocessor ⓐ running the system under study ⓑ, a large, high-resolution display ⓚ to visualize static and dynamic properties of the system, and tools for steering the computation and visualization ⓓ. In Figure 1 we show two people simultaneously interacting with PEACOCK: Wendy runs and provides input to the program being studied and Terrance steers and monitors the visualization. When the user program requires a low level of interactivity Terrance will be able to perform both tasks himself, or capture data during one run of the program and analyze it post-mortem.

PEACOCK will be able to process x86 executable files as well as Java jar files ⓕ. These can be provided directly by the user or, when historical analysis is desired, automatically extracted and built from a version control system such as CVS, using tools previously built by the PIs [35]. Static analysis tools also previously built by the PIs [36] or publicly available (for example Debray's PLTO [45] system for static analysis of x86 executables) will analyze and extract static information about the user program ⓖ. Performance data will be extracted from every level of the system under study and entered on an Event Queue ⓗ. A visualization engine ⓘ will take available data (from the Event Queue and the Static Analysis) and (using guidance from Terrance) will decide which data should be presented at any one point in time, and how it should be visualized. Publicly available 3D computer game engines and renderers such as Blender [15], Ogre [115], CaveUT [95], and X3D [22, 44] will perform the final rendering, animation, and user interaction.

The research proposed here builds on, extends, and utilizes infrastructure from previous work by the PIs. The three co-PIs have extensive background in data visualization, software visualization, performance analysis, operating systems and virtualization, manipulation of code at the native and virtual machine code levels, and in the design of large software systems:

code manipulation: PI Collberg has extensive experience in the manipulation of code at the virtual machine and binary code levels. In [27, 36–40] he describes a tool (SANDMARK) for watermarking, obfuscating, measuring, reverse engineering, and visualizing Java bytecode. SANDMARK contains a rich library for manipulating Java bytecode (control-flow analysis, data-flow analysis, slicing, etc.) which will form the basis for the Java portion of PEACOCK. In [27] the PLTO [45] x86 binary editing system is used to watermark binary executables. In [41, 42] a system for automatic extraction of semantics of machine code instructions is described.

software visualization: In [43], PI Collberg describes a system for high-quality text-and-graphic renderings of source code. We will extend these results when designing the part of PEACOCK that views and browses low-level code. In [35], PIs Kobourov and Collberg show how historical data about a program can be extracted from a CVS revision control repository, and visualized using graph-drawing techniques. The system allows easy identification of the parts of a system that have been in long-term flux and may need to be redesigned. The system also makes it possible to identify which authors have been involved with different phases of the implementation. In [58] we study the behavior of dynamically modifiable code. In [30], PIs Kobourov and Collberg describe how MIDI music can be generated from the execution of a Java program to auralize its runtime behavior. We will build on these experiences to visualize programs using PEACOCK's more natural geographic metaphor.

multi-user interaction: In [32], PIs Kobourov and Collberg describe the first non-trivial application, a multi-player version of Tetris, for the DiamondTouch hardware. DiamondTouch [47] is a touch sensitive multi-user input device developed by Mitsubishi. Follow-up work includes a study of the performance of collaborative spatial/visual tasks under different input configurations [24] and an interactive multi-user system for simultaneous graph drawing that takes advantage of direct physical interaction of several users working collaboratively [102]. We will use these experiences in designing the multi-user interaction facilities of PEACOCK.

data visualization: PI Kobourov has experience in graph drawing [18, 25, 55, 59, 67–69, 78, 79], visualization of large graphs [3, 52–54], evolving graphs [35, 58, 65, 66, 71, 72, 76, 101], graph theory [5, 13, 31, 56] and computational geometry [50, 51, 60, 61, 82]. Graph-drawing techniques and computational geometry will be important in the design of algorithms for PEACOCK's geographic visualization of code.

performance analysis: PI Hartman has published several papers on performance analysis, including SMP lock contention [87], the effect of mobile code on server performance [140], and optimizing TCP forwarder performance [141].

operating systems: PI Hartman has helped design and implement several complete operating systems including Sprite [120], Scout [108], and Joust [84,89]. He has also worked on operating system support for SMP [87], active routers [124], and PlanetLab [109]. He has done extensive research in file and storage systems, all of which involve network protocol and operating system development, including RAID-II [49], Zebra [88], Swarm [85,86,110], Gecko [7,9], and Mirage [8].

D.2.2 Data Gathering

Monitoring an interactive program's execution must be simultaneously complete and non-intrusive. On one hand, the monitoring must be detailed enough to allow proper analysis of the program. Ideally, this involves tracing all available information about the program's behavior. This information can be generated from multiple sources, including the CPU (performance monitors), an instrumented operating system, an instrumented virtual machine (such as the Java virtual machine, JVM), an instrumented program (Java bytecode, TCL scripts, binary executables), network packet sniffers, disk performance monitors, the Java profiling interface (JVMPI), etc.; see Figure 1 ⓑ. On the other hand, the monitoring must have low-overhead so as not to disturb the program being monitored. Typically this means refraining from collecting some data (potentially reducing the accuracy of the analysis), or resorting to a post-mortem analysis which does not work well for highly interactive programs.

We propose to have our cake and eat it too by throwing hardware at the problem. Hardware is inexpensive: why not use it to improve a programmer's ability to understand program behavior? With large amounts of hardware we can simultaneously simulate the program at various levels of detail, providing detailed information on program behavior without affecting overall program performance. There are several hurdles that must be overcome to make this feasible. The first is efficiently simulating the program at various levels of detail. Second, the non-determinism inherent in an interactive program must be handled so that multiple simulations produce the same results. Third, information from different components in the system and from simulations at varying levels of detail must be integrated into a complete picture of what is going on in the system. We address each of these challenges and our techniques for solving them in the following sections.

Efficient Simultaneous Simulation. Our general strategy is to run the program inside a virtual machine that allows us to collect the information necessary for analyzing the program. By using multiple virtual machines we can not only run the program at different points in its execution, we can also vary the amount of detail in the collected information. This will allow us to collect the necessary information to analyze the program without adversely affecting the program's behavior. Simulating and tracing system behavior at various levels of detail is not a new idea. SimOS [133] is a well-known system that provides this functionality. SimOS allows unmodified operating systems and applications to be run on a virtual hardware platform. The hardware platform can be simulated at various levels of detail (including transistor-level in some cases), allowing the user to make a tradeoff between detail and speed. SimOS uses Embra [150] to emulate the underlying machine. Embra uses dynamic binary translation to convert the (virtual) code that runs on the virtual machine into (real) code that runs on the real machine and has the same effect as the virtual code. This enables high-performance simulation even when the real machine has a different instruction set than the virtual machine. Higher performance is possible when the machines share the same instruction set, since most of the instructions can then be directly executed. This is the approach taken in VmWare [145]. SimOS also provides higher-detail simulation that provides more information at the cost of even slower execution.

One of the features of SimOS is a checkpoint mechanism that allows the state of the system to be recorded and used to restart subsequent simulations from the same initial conditions. This can be used to re-simulate a portion of the execution at a greater level of detail, for example. It also allows us to run multiple simultaneous simulations of the same program, using the checkpoints to provide the initial conditions.

While SimOS is a valuable tool for understanding program behavior, it has its limitations. There is a strict tradeoff between accuracy and speed that must be decided by the user. Resimulating portions of the execution at a greater level of detail is useful, but it forces the user to decide which portions of the execution to resimulate and at what level of detail, and then to wait for the results.

We propose to solve these problems by running the simulations concurrently on multiple computers. The user interacts with a primary simulation that is only detailed enough to initialize the more detailed simulations. This information includes periodic checkpoints of the program's state as well as logging I/O information so that the detailed simulations can perform I/O in exactly the same way as the primary simulation. This high-level simulation will have minimal effect on the program's performance, leading to acceptable interactive behavior.

The checkpoints produced by the primary simulation are used to initialize secondary simulations that produce information about the program at a greater level of detail. These simulations will run on additional computers so they have little effect on the primary simulation. The net result is that an interactive program can be monitored at a very detailed level while not having a significant effect on the program performance. One of the challenges for our research is efficient checkpoint creation. Checkpoint performance is not a major concern of SimOS, as a checkpoint is typically created only before the portion of the program to be studied in detail (e.g., after the OS boots).

Checkpointing a virtual machine is closely related to the issue of migrating a virtual machine: they both involve taking a consistent and complete snapshot of the virtual machine's state. Recent work on the Xen virtual machine platform [26,48] shows that virtual machine migration can be done efficiently. Xen provides *paravirtualization*, in which the virtual machine abstraction is an idealized version of the underlying real machine. This idealized virtual machine abstraction is much easier to implement than the underlying real machine, at the cost of porting the operating system and applications to the new architecture. We anticipate using binary translation (either dynamic or static) to perform this operation for applications.

Xen uses migration to allow virtual machines to move between real machines in a cluster. Their results show that this migration has a minimal impact on virtual machine performance, resulting in a 60ms service downtime during the migration. We anticipate that using similar techniques will allow us to achieve similar downtimes during checkpointing. We may be able to significantly reduce the checkpoint overhead and downtime by using a SMP computer, something that does not meet the goals for virtual machine migration in the Xen research and therefore was not considered.

Creating Determinism. Making non-deterministic programs deterministic is necessary for simultaneously simulating different portions of the program's execution. It is also an issue during debugging. Time-travel virtual machines (TTVM) [99] is a technique that allows the debugger to jump back to any point in the virtual machine's execution, as well as running it in reverse. TTVM borrows from the work on virtual machine logging and replay done in ReVirt [57] in which the virtual machine's actions are logged so that they can be replayed later. TTVM uses this logging to allow jumping back in time. Of particular importance to our work on monitoring interactive programs, TTVM logs all I/O done by the program during the initial execution and the effects of I/O are reapplied at the appropriate times during replay.

The size of the I/O log is potentially a problem, particularly if it must store all the data associated with the I/Os. The biggest concern is the hard drive, since storing all of the data written to the hard drive in the log is infeasible. TTVM avoids this problem by using a virtual hard drive. The checkpoints contain a consistent image of the hard drive, so that restarting the virtual machine from a checkpoint will cause I/Os to the hard drive to be deterministic. This assumes that no other outside process will modify the hard drive, which is true in this case, since access to the virtual drive is confined to the virtual machine. This assumption is not valid in our case, as the program being monitored may share the hard drive with other processes that are also modifying it.

We propose to solve this problem using *continuous data protection* (also called *continuous backup*), a storage industry term for fine-grain time-travel within a storage system. File system time travel has typically been achieved via snapshots: logically consistent images of the file system taken periodically (i.e. file system checkpoints) [92,105,125,135]. Recently, advances in storage capacity and snapshot algorithms have made it feasible to continuously snapshot a file system or disk [93, 148], where data is never deleted and always accessible by reverting to a previous version of the file system.

Continuous data protection greatly simplifies the checkpoint process by reducing the amount of information that must be included in the checkpoint. When starting a simulation using a checkpoint it is only necessary to present the simulation with the proper view of the file system at the time the checkpoint was taken. I/Os are still logged during the initial simulation to obtain return values and timing information, yet the actual data are not stored. During replay the information from the log is used to ensure that the I/Os are deterministic. An I/O that modifies the file system causes the view of the file system to be advanced to the point at which the I/O completed.

D.2.3 Vertical Profiling

The power of visualization is that human beings can see relationships between data that are difficult to automatically infer. For visualization to be successful it is important to present the users with enough information to be useful without overwhelming them. It is also important to do as much automated analysis as possible – there is no need to have the user discover relationships that are trivial to discern.

We refer to the analysis of data from multiple levels in the system as *vertical profiling* [90,143]. There are two aspects of the problem. First, the system must collect enough information to allow correlation between events at different levels. For example, invoking a signal handler in an application may cause a page fault. The system must trace both the signal handler and the page fault events to make finding this correlation possible. Our use of multiple simulations running on multiple processors makes collecting comprehensive information about events feasible. The user may also wish to suppress displaying correlated information. Continuing the example, if the signal handler always causes a page fault the user may wish to suppress displaying page faults caused by the signal handler so as to better visualize the remaining page faults.

A second aspect of vertical profiling is recording enough context information for an event so as to make analysis possible. This context information might be found at a different level from the one in which the event occurs. Borrowing an example from the SimOS paper [133], a page fault event is probably only useful if the currently running process is also recorded. This means that an event generated by the virtual memory system must include information obtained from the CPU scheduler. Extending the example, it might also be necessary to know which thread within the process is running, requiring information from the thread package; e.g., it might be useful to know that the page fault was caused by the garbage collector thread in the JVM.

SimOS solves this problem through *annotations*, snippets of code that are run when an event occurs to collect the proper information from the system. We will adopt a similar strategy in PEACOCK. These snippets will necessarily be specific to the system being monitored; we plan to investigate generating them automatically through static analysis of the code. We also plan to investigate whether or not the entire visualization mechanism can be implemented through annotations. In this case visualization is simply a collection of annotations that cause a change in the displayed image when an event occurs. Folding visualization into the general annotation framework is appealing, yet it is an open question as to whether or not the overheads can be kept low enough to make it feasible, even if we have multiple processors at our disposal.

D.2.4 Debugging

In PEACOCK, *low-level comprehension* (information gained from debugging) *statistical comprehension* (information gained from from profiling), *global comprehension* (information gained from static analysis), and *historical comprehension* (information gained from CVS repositories), all contribute to a complete understanding of the behavior of a program. Thus, an important goal of PEACOCK is to make debugging an integrated and inseparable task for program comprehension.

In the past, debugging has always been seen as an activity separate from other programming tasks such as design, implementation, testing, profiling, code browsing, program comprehension, etc. Only when all else fails does a programmer invoke tools such as `gdb` to locate faulty program logic. The ability to single step through a program and set breakpoints on code, events, and data, allows the programmer to follow and slowly build up an understanding of the low-level control-flow of the program.

However, many programmers can tell stories of how they found the source of a performance problem using a debugger and a logic bug using a profiler. The search for logic bugs and performance bugs are similar in nature: they require the program to be executed and artifacts of the execution analyzed by the programmer. Profiling typically gives summary statistics of the program behavior, such as the number of times a routine is called, the amount of time spent in a routine, and what percentage of total time is spent in the dynamic call-graph rooted in a particular routine. Debugging gives a more detailed understanding of the dynamic control-flow — rather than summarizing the call graph the way a profiler might, using a debugger will get you the exact call graph for a particular input.

The result of a static analysis can also be used to find logic faults. Again, taking call-graphs as an example, a static analyzer can produce a conservative estimation of a program's call graph, whereas a debugger can give the exact graph for a particular input. For example, examining static program slices can allow a programmer to find routines reachable from a particular point in a program that should not be reachable. PEACOCK will integrate the results of static analysis and debugging. For example, this will allow a programmer to ask PEACOCK to: "set a breakpoint at the entry of every function that could *possibly* be called from function `foo` and the functions it calls, recursively." Much of the required functionality (such as computing static slices) is available in the PI's SANDMARK [36] tool and Debray's PLTO [45] tool.

The integration will also be aided by the availability of large screen real-estate and many spare processing cycles. This will make it possible to overlay the static structure of a program with its dynamic behavior. For example, using our geographic metaphor, methods could be represented by city blocks, possible method calls (the result of a static call-graph analysis) by roads, actual calls (the result of actual debugging traces) by individual traffic on these roads, and performance characteristics by road type or traffic density.

We will furthermore integrate historical information in the debugging process. An important aspect of PEACOCK is the ability to extract information from a version control system. A useful feature in PEACOCK will be the ability to execute two or more closely related versions of a program in lock step, on the same user input, highlighting any differences in control-flow or data values. This *relative debugging* feature [4] will allow a programmer to run yesterday's version of a program ("where feature X was known to work") simultaneously with the current version (where X is broken), focusing on the differences in behavior between the two. Again, since PEACOCK performs extensive static analysis, static information can also be integrated in the process, visualizing simultaneously the differences between static structures and behavior. Our reliance on large multiprocessors will make interactive relative debugging possible: user and system events can be caught by a separate processor and relayed to each executing version, each running in its own virtual machine on its own processor. Our reliance on large screen real-estate will make it possible to overlay the visualization of the execution of the different versions.

To support interactivity and long-running programs, it is necessary for PEACOCK to allow programs to be executed in reverse. As described in Section D.2.2, PEACOCK monitors a running program and collects enough information to properly analyze its behavior, including running it in reverse. The general technique is to checkpoint the state of the virtual machine periodically (either on disk or in memory), then use these checkpoints to initialize the program state to just before the target instruction. The debugger then rolls the program forward until it reaches the target instruction, providing the illusion of running the program in reverse.

The interaction between debugging and monitoring is a concern when debugging a portion of a program that has not previously been executed. This "forward debugging" is an issue because the monitoring may affect the debugging and vice versa. For example, when debugging an interactive game the interaction will certainly suffer, perhaps causing the user to play the game differently than when not debugging. There appears to be no reconciliation between these two goals. Debugging requires running the program slower than normal, while proper monitoring of its behavior requires running it normally. We plan to investigate the tradeoffs between these opposing goals.

D.2.5 Visualization and Interaction

Previous work on software visualization has used simplistic ways of visualizing performance profiles and static structures of programs. These include simple tables of data, histograms, hair-ball graph drawing, etc. More regrettable is the lack of a unified model for visualizing data (static, dynamic, and historical) about a program. Since programs are inherently hierarchically structured, any metaphor must support nesting of concepts. Since visualization requirements change over time and may be particular to certain applications, the metaphor must also be “extensible,” and support many different types of visualization. We will investigate the use of a *geographic* metaphor for visualizing, browsing, and interacting with a program. The advantages are that geographic structures are inherently hierarchical (buildings occur within city blocks within burrows within cities within countries within continents), are well understood, and that there are a multitude of tools for manipulating geographic data. We plan to investigate the use of *computer game engines* to represent, visualize, browse, and interact with the program under study. Many open source game engines are available and most software engineers will have much experience interacting with computer games, and thus the idea of browsing code by flying through a landscape will not appear alien to them.

Browsing, debugging, profiling, and tracing code allow users to interact with the program at different levels of detail. We integrate these levels into one comprehensive model. Just like an airplane has a pilot and a co-pilot, we will allow multiple users to interact with the same program. This will become necessary when the program under study is so fast-paced that one user will need to pay complete attention to interacting with it, freeing up the second user to monitor the state of the computation. We will draw on our experience with new hardware for multi-user interaction [24, 32, 102].

Visualizing Static Structures. Much research has been done within the software visualization community to visualize different aspects of source code. This includes static call graphs, inheritance graphs, control-flow graphs, package graphs, identifier cross-references, program slices, data flow, etc. PI Collberg’s SANDMARK system for code obfuscation and software watermarking, for example, uses standard graph-drawing techniques to view control-flow graphs and inheritance graphs, in order to allow simulated manual attacks against software watermarks. Standard visualization techniques become impractical when the structures to be visualized grow large. In particular, drawings of graphs with more than a few hundred vertices fail to convey much information, except when the underlying structures are very simple (trees) or regular (meshes). Building on earlier work on visualization of large graphs [78, 79] we will explore the use of compound-fisheye views [3] and multi-dimensional, multi-scale techniques [52, 53].

In Section D.2.1, Scenario 1, Wendy uses PEACOCK for a tour of a legacy system with which she needs to familiarize herself. There is a multitude of relationships between code structures for Wendy to explore, and a multitude of ways to visualize these relationships using our geographic metaphor. The goal of this part of the project is to explore different uses of the metaphor to find out which layouts give the best visualization experience, and to develop algorithms for realizing these layouts. As an explicit example, consider the problem of simultaneously visualizing a Java package graph, inheritance graph, and static class call graph. All three graphs share the same set of vertices (classes). Edges in the call graph correspond to classes whose methods call each other. Edges in the inheritance graph correspond to the inheritance relation. The package graph is different in that it is a containment hierarchy. We would like to embed the vertices (classes) in the terrain so as to take into consideration all three relationships. To capture the two relationships defined by the inheritance and call graphs we can adopt our simultaneous graph embedding techniques [18, 67, 70] to the geographic metaphor. Generic graph visualization algorithms attempt to create layouts where distances between vertices in the layout correspond to the graph theoretic distances in the abstract graph. Algorithms for simultaneous graph embedding extend this notion to multiple graphs on the same set of vertices. Thus, we can obtain layouts where vertex locations are determined by more than one abstract graph.

While we can use both the inheritance and call graphs in obtaining an initial graph layout, the package graph contains more than just the classes. The package graph is a tree whose vertices are classes and supernodes (multiple classes). Thus, we need a different approach in order to ensure that the package graph influences the layout. We propose to realize the package graph as continents and countries (see panel ⓒ, for example, in Figure 2), by adapting a treemap approach to our geographic metaphor. Treemaps [139] provide efficient space-filling layouts of tree-like structures. The vertices of the tree are displayed as nested rectangles in the treemap. The children of a node are within the rectangle of the parent. Squarified treemaps ensure good aspect ratio for the rectangles [21] and ordered treemaps keep related items spatially close to each other in the map [12]. Adapting the treemap approach to the geographic metaphor would allow us to restrict vertices of the graph to specific regions on the terrain. After restricting the movement of vertices to their corresponding regions (defined by the package hierarchy) we can apply the simultaneous embedding technique using the inheritance and call graph relations. Thus we can capture the containment relation from the package graph and also capture the relations from the inheritance and call graphs.

PEACOCK will process Java jar files with SANDMARK [36] to extract any information about the code that could be visualized. PLTO [45] will be used to extract similar data from x86 executables. A visualization engine (Figure 1 ⓘ) decides which data should be presented based on user guidance. The final step (Figure 1 ⓙ) is to render appropriate geographic scenes and animation, and supporting user interactivity such as zooming, fly-throughs, etc. There are a large number of sophisticated open source tools designed to support computer game development that PEACOCK will use to implement this stage. Examples include Blender [15], Ogre [115], CaveUT [95] (a virtual reality cave), FlightGear [75] (an open source multi-display flight simulator), OpenGC [116] (an open source glass cockpit), X3D [22,44] (an XML-based virtual reality system), and Terragen [144] (a photorealistic scenery rendering program).

Visualizing Historical Development. In Section D.2.1, Scenario 2, Terrance's code is visualized from a historical perspective using PEACOCK. The natural way of doing this is to employ animation of static code layouts. In Figure 2 we show how the historical development of a Java program can be visualized geographically. In ⓐ, the program starts out as one class `class1` (a city) within a package `package1` (a country). A continent represents the entire program. In ⓑ, the program has developed into three classes, each within its own country. In ⓒ, `class1` now resides within the package `package1.package4`, and by slight abuse of the geographic metaphor we visualize this as a country within a country[2].

Our previous work on visualizing the historical development of a program [35] extracted a Java program (SANDMARK, currently 120,000 lines of Java code) from its CVS database. One version was extracted per day, the program was built (compiled to Java bytecode), and control-flow graphs, inheritance graphs, and static call-graphs were constructed. These graphs were then visualized using a temporal graph-drawing system [76]. Tree-structures (such as inheritance and package graphs) will be easy to visualize geographically, as illustrated in Figure 2. Static call graphs can be laid out by using standard graph-drawing techniques: classes (cities) contain methods, methods (city blocks) are connected by call edges (roads, waterways), and cities are located to minimize distance between tightly coupled classes.

In this scenario, the underlying problem is visualization of changing graphs. *Evolving graph visualization* is the off-line version of the problem, where we know in advance all of the changes that the graph will undergo, e.g., day-by-day inheritance graphs. *Dynamic graph visualization* is the on-line version where we do not know what changes will occur, e.g. call-graphs of an executing program in real time. There are two important layout criteria to consider: the *readability* of the individual layouts and the *mental map preservation* in the series of drawings. The readability of individual drawings depends on aesthetic criteria such as display of symmetries, uniform edge lengths, and minimal number of crossings. Preservation of the mental map can be achieved by ensuring that vertices and edges that appear in consecutive graphs in the series, remain unchanged. These two criteria are often contradictory. If we obtain individual layouts

[2]While unusual, countries within countries do exist. Examples include Lesotho within South Africa, and the Hopi reservation within the Navajo reservation within Arizona within the United States.

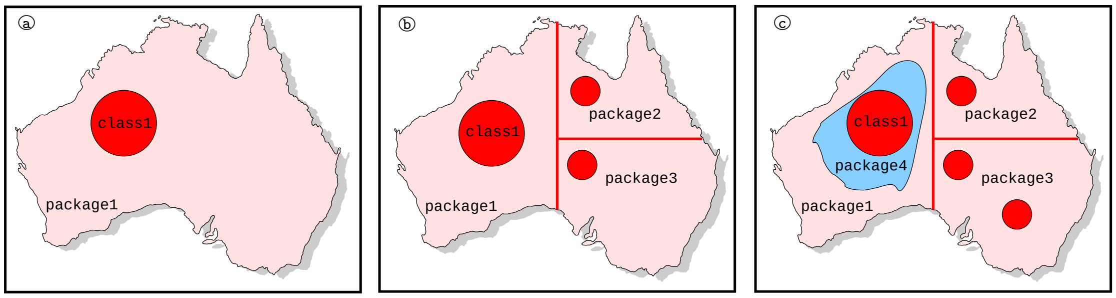

Figure 2: Visualizing the historical development of a program.

for each graph, without regard to other graphs in the series, we may optimize readability at the expense of mental map preservation. Conversely, if we fix the common vertices and edges in all graphs once and for all, we are optimizing the mental map preservation though the individual layouts may be far from readable. Thus, we can measure the effectiveness of various approaches for visualization of evolving and dynamic graphs by measuring the readability of the individual drawings, and the overall mental map preservation. We expect to use animation and morphing to show changes in the code over time while at the same time aiding in the mental map preservation. For example, a package that is refactored into two can be animated as a country being split into two, with certain cities (classes) falling into the hands of the different countries.

Visualizing Performance Profiles. In Section D.2.1, Scenario 3, Philip used PEACOCK to visualize performance. Our goal is to be able to overlay this information on the static visualization of the code as described above. As shown in Figure 1 ⓚ, a Programmer's Cockpit can visualize performance in two ways. First, global performance levels are shown as a collection of dials and charts that give Philip a quick overview of the health of the system. Current page fault rate, amount of freed memory after garbage collections, number of active threads, current CPU utilization, are examples of such information. Localized performance data, (information directly tied to a particular segment of code) we anticipate visualizing geographically.

In this case, we are concerned with the visualization of truly dynamic graphs, that change in real-time. One of these graphs that is notoriously difficult to deal with, yet can aid greatly in the discovery of dynamic allocation patterns, is the heap-graph. All programmers are familiar with "pointer-bugs," "memory leaks," and "heap fragmentation." The reason why such problems occur is not only due to programmer incompetence. There are, in fact, fundamental theoretical reasons why reasoning about pointer-based data-structures is difficult. It is known that determining whether two pointers in a program will ever point to the same heap object (the "aliasing problem") is undecidable in general [128]. Similarly, determining whether a program that constructs a dynamic data-structure will actually build a list, a tree, a DAG, or a graph, is also, in general, undecidable [81]. In long-running programs the heap can contain millions of objects which makes visualization a formidable task. Furthermore, the heap changes frequently which means that there may be much activity between time-slices that needs to be presented to the user.

Visualizing the heap graph can also be combined with visualization of the call-graph and the inheritance graph, to dampen the effect of the large number of changes that occur in the heap graph. In addition, there are many possible ways to explore the geographic metaphor: activity levels can be expressed as elevation of cities (classes), as the height of buildings (methods), as the amount of light given off by a city, or as the traffic flow on a highway (method call interconnect).

D.2.6 Related Work

Software Visualization. Software visualization aim to improve human understanding of computer programs by portraying them in a form that is more readable than mere source code [111, 112, 127, 132].

BALSA [19] and Zeus [20] annotate a program with hooks so that "interesting events" such as changes to data structures and subroutine calls and returns can be relayed to the visualization system. SHriMP [142] shows inheritance hierarchies and aggregation using a variety of graphical views. Young [151] visualizes dense call-graphs as stacks of cubes, viewing the result in a virtual reality environment.

Many tools visualize historical information stored by change management systems such as CVS [10, 11, 23, 62, 96, 106, 146]. SoftChange [106] produces textual reports of the complexity, size, purpose and author of program changes. Ball [10, 11] describes tools that visualizes many different aspects of software using line, pixel and hierarchical representations. Eick [62] visualizes software changes using bar-graphs, pie-charts, matrix views, and cityscape views.

Integrated systems for program understanding [63,97,117,118,149] include RevEngE [149] (support for pattern matching, redundancy analysis, and reverse engineering), Gammatella [97, 118] (storing and visualizing program-execution data using a treemap view), and the "War Room Command Console" [117] (eight linked consoles to present system development information). Eng [63] proposes a visualization framework to provide static and dynamic program visualizations.

Tools for visualizing dynamic behavior or profiling information [17, 46, 80, 122, 129–131, 147, 152] include EVolve [147] (visualization of the runtime behavior of Java programs using barcharts and dot-plots), OverView [46] (an event-based Eclipse plug-in for runtime visualization of distributed systems), Desert [129] (customizable visualizations of dynamic call counts, dynamic call traces, load and store traces), and FileVis [152] (high-level overviews of a software system using 3D graphics and VR technology).

Aesthetic computing [1,74,126] addresses artistic issues in visualization. The RUBE system [74] allows for model visualization with different metaphors (e.g., event graphs visualized using a cityscape metaphor). The RelVis system [126] uses Kiviat diagrams using a tree-ring visualization to provide integrated graphical views of source code and release history.

Little academic research has utilized software developed within the computer game community. CaveUT [95] uses the *Unreal Engine* to construct immersive virtual reality environments. Knight and Munro [100] describe using the *Quake 2* game engine to create 3D visualizations of source code.

Profiling. There are two principal methods of collecting profiling information. Code instrumentation approaches such as `gprof` [83] insert instructions at compile-time or post-link-time, counting procedure invokations, basic block executions, branches taken, etc. Sampling-based systems such as the PCT system [14] periodically suspend the program and examine the program state. PCT's basic philosophy is similar to this proposal, in that profiling is viewed as a form of debugging.

The Java virtual machine profiling interface (JVMPI [2]) is used by the Pajè [119] and JaViz [98] systems to collect traces of distributed systems that are then visualized (for example, as call-graphs) off-line.

To correctly capture the performance of a system execution information has to be collected at multiple levels (Figure 1 ⓑ). Sweeney, Hauswirth, et al. [90, 143] integrate information from an instrumented JVM with data from hardware performance counters, allowing performance to be visualized off-line.

Debugging. Early work on debugging is surveyed by Paxon [123] and McDowell *et al.* [104,123]. There are four basic implementation techniques to enable reverse debugging: *logging*, *checkpointing*, *forking*, and *instrumentation*. *Logging* was introduced by Zelkowitz [153], in the first reverse execution system. Code is added to the binary (by modifying the compiler) to record all changes to variables, resulting in a 40% code expansion. Moher's PROVIDE [107] similarly logs variable updates but was limited to short-running programs due to the rapid growth of the log. Netzer [114] compresses the log at the cost of less flexible backward motions. Feldman and Brown's IGOR [73] system uses *checkpointing* to write program state (modified pages and open file descriptors) to disk at fixed intervals. The overhead ranges from 40 to 380%. Pan and Linton's Recap [121] system periodically *forks* off a child process that holds the state of the program at that point. If `fork` is implemented using copy-on-write, this can be an efficient approach. Booth [16] uses a similar idea but also periodically thins out the forked processes in a log-like fashion, reducing the memory requirement while making short backwards time-travel fast. In addition to checkpointing by forking, the binary is instrumented with counters that make the reverse analogues of `step`, `next`, `continue` efficient. Backwards execution also requires deterministic re-execution. Booth [16] stores the return values of system calls and replays them during re-execution. Recap [121] stores the times of system calls, signals, and shared memory accesses, enforcing deterministic re-execution.

Much research has been done on parallel debugging. Fowler et al. [77] describe a system for debugging and performance tuning on shared memory multiprocessors. They use a two-phased approach where the program under study is first run under monitoring and each process stores an execution trace to disk. In a second off-line phase the traces are merged, visualized, analyzed for performance problems, or analyzed for problems due to incorrect accesses to shared variables. The execution history is coarse-grained — only synchronization information is stored — and hence relatively inexpensive to capture. Based on the synchronization information, programs can also be replayed deterministically.

Abramson and Sosič [4] present a system for *relative debugging*. The idea is to run a version of a program known to work in parallel with a new, buggy, version and display the differences in the data structures.

D.3 Development Plan, Evaluation, and Broader Impacts

D.3.1 Development Plan

The proposed research will be carried out over a three-year period. It involves a significant amount of design, development, and evaluation. We request support for two PhD-level graduate students and one post-doc to assist the three PIs in carrying out this research. We anticipate the following development plan:

Year 1: During the first year we will focus on static analysis (Scenario 1 in Section D.2.1). We will develop a prototype system that uses a game engine to enable static analysis on a large-scale display. We will also begin research on historic and dynamic analysis, specifically developing the infrastructure for running multiple simultaneous simulations on multiple virtual machines. This will primarily focus on checkpoint and continuous data protection technologies.

Year 2: During the second year we will extend our work on static analysis to include historic analysis (Scenario 2 in Section D.2.1). We will develop better visualization technologies based on our results from the first year, as well as develop a prototype dynamic visualization system. We will also extend our virtual machine infrastructure to include vertical profiling support and improve the overall infrastructure based on our results from the first year.

Year 3: During the third year we will integrate static, historic, and dynamic analysis into a single integrated tool. We will complete our work on simultaneous multi-level simulation and visualization.

D.3.2 Evaluation

The goal of the proposed work is to answer two questions: (1) *Can software be visualized more effectively using metaphors borrowed from computer games, if massive high-resolution screen real-estate is available?* (2) *Can software be more thoroughly analyzed statically, dynamically, and historically if massive computational resources are available?* These questions can be reformulated as "will the integrated environment provided by PEACOCK allow programmers to find logic and performance bugs more effectively than currently available tools?" and "how much hardware resources are necessary to collect, store, analyze, and visualize data about a program while maintaining an adequate interactive experience for the programmer?" We will collaborate with psychologists in designing and performing effective user studies to answer these questions.

The first question could be addressed by a study in which we vary the visualization methods and measure their effectiveness by keeping track of the time users need to solve a particular problem. The second question could be addressed by varying the amount of resources available to PEACOCK, while running test scenarios similar to those in Section D.2.1.

D.3.3 Broader Impacts

The work will be carried out by the investigators, graduate students, and undergraduate students. Undergraduates have played key roles in our recent projects, as discussed in Section D.4, and we plan to continue employing them. We anticipate that many undergraduates with extensive gaming experience will be able to actively contribute to the research problems as well as to the implementation of the system. The virtual worlds created by the computer game industry have become powerful metaphors well understood by a large segment of the population and this project aims to extend their use to the software visualization domain.

This project will foster the development and education of the participating students and will influence the curriculum development of the PIs. We will also make all software developed as part of the project available to the broader research community.

D.4 Results from Prior NSF Support

D.4.1 Collberg

Collberg is PI on NSF grant CCR-0073483, entitled *Code Obfuscation, Software Watermarking, and Tamper-Proofi ng —Tools for Software Protection*, $265,000, September 1, 2000–August 31, 2004. This project resulted in papers describing techniques for constructing error-correcting graphs [28], an overview of software protection techniques [37], a description of the SANDMARK software protection infrastructure [36], novel software watermarking algorithms [27], SANDMARK's obfuscation executive [91], and evaluation of published software protection techniques [29, 113, 134]. Three bachelor's and one master's thesis have been completed, and one PhD thesis is expected to complete this year.

D.4.2 Kobourov

Kobourov is PI on NSF grant ACR-022920, entitled *Visualization of Giga-Graphs and Graph Processes*, for the period September 2002 through August 2005, $240,358. This project has resulted in a number of publications on visualization of large graphs and graphs that evolve through time [18, 35, 58, 66–69, 76] and several software systems for graph visualization, including GraphAEL [65] (for visualization of computing literature) and GMorph [71] (for intersection-free morphing of planar graphs). Many students have been involved in this research, both of the graduate and undergraduate levels. Cesim Erten co-authored ten papers [18, 59, 65–68, 70–72, 94] and completed a PhD on this topic in 2004 [64]. Four MS students were co-authors on papers related to the project [3, 24, 35, 65, 66, 71, 72, 76, 102, 103]. Ten of our publications have at least one undergraduate co-author [24, 31–35, 58, 65, 66, 70, 76].

D.4.3 Hartman

Hartman is PI on NSF grant CCR-9624845, entitled *A Caching Infrastructure*, $200,000, 1996–2000. This project resulted in the publication of several papers on caching and distributed storage systems. Research on cooperative caching resulted in two papers [137, 138] and one Ph.D. thesis [136]. Research on distributed storage systems resulted in six papers [7, 9, 85, 86, 110, 140] and one Ph.D. thesis [6]. One paper was also published on efficient network proxy implementation [141]. We also made the Gecko Web proxy and Swarm storage system software available to the broader research community. Two Ph.D. theses and two Master's theses directly resulted from this research grant.

E References

[1] Aesthetic computing. Dagstuhl Seminar #2291, 2003.

[2] Java virtual machine profiler interface. *IBM Systems Journal*, 39(1), 2000.

[3] J. Abello, S. G. Kobourov, and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In *12th Symposium on Graph Drawing (GD)*, pages 431–442, 2004.

[4] D. Abramson and R. Sosic. Relative debugging using multiple program versions. In *8th Int. Symp. on Languages for Intensional Programming*, Sydney, May.

[5] B. Awerbuch and S. G. Kobourov. Polylogarithmic-overhead piecemeal graph exploration. In *Proceedings of the 11th Annual ACM Conference on Computational Learning Theory (COLT)*, pages 280–286, 1998.

[6] S. Baker. *Virtualized File Service*. PhD thesis, University of Arizona, 2005. Expected completion in Fall 2005.

[7] S. Baker and J. H. Hartman. The design and implementation of the Gecko NFS Web proxy. *Software–Practice and Experience*, 31(7):637–665, 2001.

[8] S. Baker and J. H. Hartman. The Mirage NFS router. In *Proceedings of the 29th IEEE Conference on Local Area Networks*, Tampa, FL, Nov. 2004.

[9] S. M. Baker and J. H. Hartman. The Gecko NFS Web proxy. *Computer Networks*, 31(11–16):1724–1736, 1999.

[10] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[11] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *ICSE ’97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[12] B. Bederson, B. Shneiderman, and M. Wattenberg. Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, 2002.

[13] T. Biedl, E. Demaine, C. A. Duncan, R. Fleischer, and S. G. Kobourov. Tight bounds on maximal and maximum matching. *Journal of Discrete Mathematics*, 285(1):7–15, 2004.

[14] C. BLAKE and S. BAUER. Simple and general statistical profiling with pct. In *Usenix Technical Conference*, 2002.

[15] Blender. 3d modeling, animation, rendering, post-production, interactive creation and playback. `http://www.blender3d.com/cms/Home.2.0.html`, 2005.

[16] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310. ACM Press, 2000.

[17] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: a flexible environment for computer systems visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73, 2000.

[18] P. Brass, E. Cenek, C. A. Duncan, A. Efrat, C. Erten, D. Ismailescu, S. G. Kobourov, A. Lubiw, and J. S. B. Mitchell. On simultaneous graph embedding. *Computational Geometry: Theory and Applications*. To appear in 2005. (Preliminary version in *8th Workshop on Algorithms and Data Structures (WADS)*, p. 243-255, 2003).

[19] M. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36, 1988.

[20] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. Technical Report 75, 28 1992.

[21] M. Bruls, K. Huizing, and J. J. van Wijk. Squarified treemaps. In *Proc. Joint Eurographics/IEEE TVCG Symp. Visualization, VisSym*, pages 33–42, 2000.

[22] L. Bullard. Extensible 3d: Xml meets vrml. *O'Reilly XML.com*, Aug. 2003. `http://www.xml.com/pub/a/2003/08/06/x3d.html`.

[23] M. Burch, S. Diehl, and P. W. gerber. Visual data mininng in software archives. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 37–46, St. Louis, Missouri, USA, May 2005.

[24] J. Cappos, S. G. Kobourov, M. Miles, M. Stepp, K. Pavlou, and A. Wixted. Collaboration with diamondtouch. In *10th International Conference on Human Computer Interaction (INTERACT)*. To appear in 2005.

[25] C. C. Cheng, C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Drawing planar graphs with circular arcs. *Discrete and Computational Geometry*, 25:405–418, 2001.

[26] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[27] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, June 2004. (To appear).

[28] C. Collberg, E. Carter, S. Kobourov, and C. Thomborson. Error-correcting graphs. In *Workshop on Graphs in Computer Science (WG'2003)*, June 2003.

[29] C. Collberg, A. Huntwork, E. Carter, and G. Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. Technical Report TR04-06, Department of Computer Science, University of Arizona, 2004.

[30] C. Collberg, S. Kobourov, C. Hutcheson, J. Trimble, and M. Stepp. Monitoring java programs using music. Technical Report TR05-04, Department of Computer Science, University of Arizona, 2005.

[31] C. Collberg, S. G. Kobourov, E. Carter, and C. Thomborson. Error-correcting graphs for software watermarking. In *29th Workshop on Graph Theoretic Concepts in Computer Science (WG)*, pages 156–167, 2003.

[32] C. Collberg, S. G. Kobourov, S. Kobes, B. Smith, S. Trush, and G. Yee. Tetratetris: An application of multi-user touch-based human-computer interaction. In *9th International Conference on Human-Computer Interaction (INTERACT)*, pages 81–88, 2003.

[33] C. Collberg, S. G. Kobourov, J. Louie, and T. Slattery. SPLAT: A system for self-plagiarism detection. In *Proceedings of the IADIS Conference WWW/Internet*, pages 508–514, 2003.

[34] C. Collberg, S. G. Kobourov, J. Miller, and S. Westbrook. Algovista: A tool to enhance algorithm design and understanding. In *7th Annual Symposium on Innovation and Technology in Computer Science Education (ITICSE)*, pages 228–228, 2002.

[35] C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *ACM Symposium on Software Visualization (SoftVis)*, pages 77–86, 2003.

[36] C. Collberg, G. Myles, and A. Huntwork. SandMark — a tool for software protection research. 1(4):40–49, July-August 2003.

[37] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, The Department of Computer Science, University of Arizona, Feb. 2000.

[38] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 IEEE International Conference on Computer Languages*, pages 28–38.

[39] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

[40] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. 25th. ACM Symposium on Principles of Programming Languages (POPL 1998)*, pages 184–196, Jan. 1998.

[41] C. S. Collberg. Reverse interpretation + mutation analysis = automatic retargeting. In *Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997. SIGPLAN, ACM. www.cs.auckland.ac.nz/ collberg/Research/Publications/Collberg97b/index.html.

[42] C. S. Collberg. Automatic derivation of compiler machine descriptions. *ACM Trans. Program. Lang. Syst.*, 24(4):369–408, 2002.

[43] C. S. Collberg, S. Davey, and T. A. Proebsting. Language-agnostic program rendering for presentation, debugging and visualization. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 183, Washington, DC, USA, 2000. IEEE Computer Society.

[44] W. D. Consortium. X3d. `http://www.web3d.org`, 2005.

[45] S. Debray, B. Schwarz, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Rewriting (WBT-2001)*, Sept. 2001.

[46] T. Desell, H. N. Iyer, C. A. Varela, and A. Stephens. Overview: A framework for generic online visualization of distributed systems. *Electr. Notes Theor. Comput. Sci.*, 107:87–101, 2004.

[47] P. H. Dietz and D. Leigh. Diamondtouch: A multi-user touch technology. In *Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01)*, pages 219 – 226, Nov. 2001.

[48] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[49] A. L. Drapeau, K. W. Shirrif, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, D. A. Patterson, E. K. Lee, P. H. Chen, and G. A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 234–244, 1994.

[50] C. Duncan, D. Eppstein, and S. G. Kobourov. The geometric thickness of low degree graphs. In *20th ACM Symposium on Computational Geometry*, pages 340–346, 2004.

[51] C. A. Duncan, A. Efrat, S. G. Kobourov, and C. Wenk. Drawing with fat edges. In *9th Symposium on Graph Drawing (GD)*, pages 162–177, 2001.

[52] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees and their use for drawing large graphs. *Journal of Graph Algorithms and Applications*, 4:19–46, 2000.

[53] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *Journal of Algorithms*, 38:303–333, 2001.

[54] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Planarity-preserving clustering and embedding for large planar graphs. *Computational Geometry: Theory and Applications*, 24(3):203–224, 2002.

[55] C. A. Duncan and S. G. Kobourov. Polar coordinate drawing of planar graphs with good angular resolution. *Journal of Graph Algorithms and Applications*, 7(4):311–333, 2003.

[56] C. A. Duncan, S. G. Kobourov, and V. S. A. Kumar. Optimal constrained graph exploration. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 807–814, 2001.

[57] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[58] B. Dux, A. Iyer, S. Debray, D. Forrester, and S. G. Kobourov. Visualizing the behaviour of dynamically modifiable code. In *13th IEEE Workshop on Porgram Comprehension*, pages 337–340, 2005.

[59] A. Efrat, C. Erten, and S. G. Kobourov. Fixed-location circular-arc drawing of planar graphs. In *11th Symposium on Graph Drawing (GD)*, pages 147–158, 2003.

[60] A. Efrat, S. Kobourov, and A. Lubiw. Computing homotopic shortest paths efficiently. *Computational Geometry: Theory and Applications*. To appear in 2005. (Preliminary version in *10th Annual Symposium on Algorithms (ESA)*, p.411-423, 2002).

[61] A. Efrat, S. G. Kobourov, M. Stepp, and C. Wenk. Growing fat graphs. In *18th Annual Symposium on Computational Geometry (SCG)*, pages 277–278, 2002.

[62] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *Software Engineering*, 28(4):396–412, 2002.

[63] D. Eng. Combining static and dynamic data in code visualization. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–50, New York, NY, USA, 2002. ACM Press.

[64] C. Erten. *Simultaneous Embedding and Visualization of Graphs*. PhD thesis, University of Arizona, 2004.

[65] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. GraphAEL: Graph animations with evolving layouts. In *11th Symposium on Graph Drawing (GD)*, pages 98–110, 2003.

[66] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Exploring the computing literature using temporal graph visualization. In *Proceedings of Visualization and Data Analysis (VDA)*, pages 45–56, 2004.

[67] C. Erten and S. G. Kobourov. Simultaneous embedding of planar graphs with few bends. In *12th Symposium on Graph Drawing (GD)*, pages 195–206, 2004.

[68] C. Erten and S. G. Kobourov. Simultaneous embedding of a planar graph and its dual on the grid. *Theory of Computing Systems*, 38(3):313–327, 2005.

[69] C. Erten, S. G. Kobourov, A. Navabia, and V. Le. Simultaneous graph drawing: Layout algorithms and visualization schemes. *Journal of Graph Algorithms and Applications*. To appear in 2005. (Preliminary version in *11th Symposium on Graph Drawing (GD)*, p. 437-449, 2003).

[70] C. Erten, S. G. Kobourov, A. Navabia, and V. Le. Simultaneous graph drawing: Layout algorithms and visualization schemes. In *11th Symposium on Graph Drawing (GD)*, pages 437–449, 2003.

[71] C. Erten, S. G. Kobourov, and C. Pitta. Intersection-free morphing of planar graphs. In *11th Symposium on Graph Drawing (GD)*, pages 320–331, 2003.

[72] C. Erten, S. G. Kobourov, and C. Pitta. Morphing planar graphs. In *20th ACM Symposium on Computational Geometry (SCG)*, pages 320–331, 2004.

[73] S. I. Feldman and C. B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, New York, NY, USA, 1988. ACM Press.

[74] P. A. Fishwick, J. Lee, M. Park, and H. Shim. Next generation modeling i: Rube: a customized 2d and 3d modeling framework for simulation. In *Winter Simulation Conference*, pages 755–762, 2003.

[75] Flightgear. `http://www.flightgear.org`, 2005.

[76] D. Forrester, S. G. Kobourov, A. Navabi, K. Wampler, and G. Yee. graphael: A system for generalized force-directed layouts. In *12th Symposium on Graph Drawing (GD)*, pages 454–466, 2004.

[77] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis onlarge-scale multiprocessors. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 163–173, New York, NY, USA, 1988. ACM Press.

[78] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. *Computational Geometry: Theory and Applications*, 29(1):3–18, 2004.

[79] P. Gajer and S. G. Kobourov. GRIP: Graph dRawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.

[80] P. V. Gestwicki and B. Jayaraman. Methodology and architecture of jive. In *SOFTVIS*, pages 95–104, 2005.

[81] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL'96*, pages 1–15, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[82] H. González-Baños, A. Efrat, S. G. Kobourov, and L. Palaniappan. Optimal motion strategies to track and capture a predictable target. In *IEEE Conference of Robotics and Automation (ICRA)*, pages 411–423, 2003.

[83] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[84] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Experiences building a communication-oriented JavaOS. *Software: Practice & Experience*, 30(10):1107–1126, 2000.

[85] J. H. Hartman, S. Baker, and I. Murdock. Customizing the Swarm storage system using agents. *Software–Practice and Experience*. To appear.

[86] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm scalable storage system. In *International Conference on Distributed Computing Systems*, pages 74–81, 1999.

[87] J. H. Hartman and J. K. Ousterhout. Performance measurements of a multiprocessor Sprite kernel. In *USENIX Summer*, pages 279–288, 1990.

[88] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[89] J. H. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A platform for communications-oriented liquid software. *IEEE Computer*, 32(4):50–56, Apr. 1999.

[90] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[91] K. Heffner and C. Collberg. The obfuscation executive. Technical Report TR04-03, Department of Computer Science, University of Arizona, 2004.

[92] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. pages 235–246, January 1994.

[93] C. B. M. III and D. Grunwald. Peabody: The time travelling disk. In *In 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2003.

[94] A. Iyer, A. Efrat, C. Erten, D. Forrester, and S. G. Kobourov. A force-directed approach to sensor localization. In *13th Symposium on Graph Drawing (GD)*. To appear in 2005.

[95] J. Jacobson. Using "caveut" to build immersive displays with the unreal tournament engine and a pc cluster. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 221–222, New York, NY, USA, 2003. ACM Press.

[96] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE*, pages 360–370, 1997.

[97] J. A. Jones, A. Orso, and M. J. Harrold. Gammatella: visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[98] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja, and P.-C. Yew. JaViz: A client/ server Java profiling tool. *IBM Systems Journal*, 39(1):96–117, 2000.

[99] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[100] C. Knight and M. Munro. Using an existing game engine to facilitate multi-user software visualisation. Technical Report 8/98, Visualisation Research Group, Centre for Software Maintenance, Department of Computer Science, University of Durham, 1998.

[101] S. G. Kobourov and M. Landis. Morphing planar graphs in spherical space. In *13th Symposium on Graph Drawing (GD)*. To appear in 2005.

[102] S. G. Kobourov and C. Pitta. An interactive multi-user system for simultaneous graph drawing. In *12th Symposium on Graph Drawing (GD)*, pages 492–503, 2004.

[103] S. G. Kobourov and K. Wampler. Non-Euclidean spring embedders. *IEEE Transactions on Visualization and Computer Graphics*. To appear in 2005. (Preliminary version in *10th Annual IEEE Symposium on Information Visualization (INFOVIS)*, p. 207-214, 2004).

[104] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[105] I. Microsoft. Windows XP system restore. `http://msdn.microsoft.com/library/default.asp?URL=/library/techart/windowsxpsystemrestore.htm`.

[106] A. Mockus, S. Eick, T. Graves, and A. Karr. On measurement and analysis of software changes, 1999.

[107] T. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[108] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the OSDI*, pages 153–168, Oct. 1996.

[109] S. Muir, L. Peterson, M. Fiuczynski, J. Cappos, and J. Hartman. Proper: Privileged operations in a virtualised system environment. In *Proceedings of the 2005 Usenix Technical Conference*, 2005.

[110] I. Murdock and J. H. Hartman. Swarm: A log-structured storage system for Linux. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference*, June 2000.

[111] B. A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 59–66, Apr. 1986.

[112] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, Mar. 1990.

[113] G. Myles and C. Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *International Conference on Information Security and Cryptology*, 2003.

[114] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 313–325, New York, NY, USA, 1994. ACM Press.

[115] Ogre. 3d rendering engine. `http://www.ogre3d.org`, 2005.

[116] The open source glass cockpit project. `http://www.opengc.org`, 2005.

[117] C. O'Reilly, D. W. Bustard, and P. J. Morrow. The war room command console: shared visualizations for inclusive team coordination. In *SOFTVIS*, pages 57–65, 2005.

[118] A. Orso, J. A. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *SOFTVIS*, pages 67–76, 211, 2003.

[119] F.-G. Ottogalli, C. Labbé, V. Olive, B. de Oliveira Stein, J. Chassin de Kergommeaux, and J.-M. Vincent. Visualisation of distributed applications for performance debugging. In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Kenneth Tan, editors, *ICCS'01: International Conference in Computational Science*, Lecture Notes in Computer Science 2074, pages 831–840, Berlin, Heidelberg, 2001. Springer.

[120] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[121] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 124–129, New York, NY, USA, 1988. ACM Press.

[122] W. D. Pauw, D. H. Lorenz, J. M. Vlissides, and M. N. Wegman. Execution patterns in object-oriented visualization. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, 1998.

[123] V. Paxson. A survey of support for implementing debuggers. Fall Semester 1990.

[124] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, Mar. 2001.

[125] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[126] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *SOFTVIS*, pages 67–75, 2005.

[127] B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[128] G. Ramalingam. The undecidability of aliasing. *ACM TOPLAS*, 16(5):1467–1471, Sept. 1994.

[129] S. P. Reiss. Software visualization in the desert environment. In *PASTE*, pages 59–66, 1998.

[130] S. P. Reiss and M. Renieris. The bloom software visualization system. In *Software Visualization – From Theory to Practice*. MIT Press, 2003.

[131] S. P. Reiss and M. Renieris. Jove: java as it happens. In *SOFTVIS*, pages 115–124, 2005.

[132] G.-C. Roman and K. C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, 1993.

[133] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM TRansactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[134] T. R. Sahoo and C. Collberg. Software watermarking in the frequency domain: Implementation, analysis, and attacks. Technical Report TR04-07, Department of Computer Science, University of Arizona, 2004.

[135] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[136] P. Sarkar. *Hint-based cooperative caching*. PhD thesis, University of Arizona, 1998.

[137] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceeding of the 2nd ACM Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, 1996.

[138] P. Sarkar and J. H. Hartman. Hint-based cooperative caching. *ACM Transactions on Computer Systems*, 18(4):387–419, 2000.

[139] B. Shneiderman. Tree visualization with treemaps: a 2-d space-filling approach. Technical report, Human-Computer Interaction Lab, University of Maryland, 1991.

[140] T. Spalink, J. H. Hartman, and G. Gibson. A mobile agent's effect on file service. *IEEE Concurrency*, 8(2):62–69, 2000.

[141] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, 2000.

[142] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Muller. On integrating visualization techniques for effective software exploration. pages 38–45, 1997.

[143] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In *3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.

[144] terragen — photorealistic scenery rendering software. `http://www.planetside.co.uk/terragen/tgd/tgd.shtml`, 2005.

[145] VMware, Inc. VMware virtual machine technology. `http://www.vmware.com`, 2005.

[146] L. Voinea, A. Telea, and J. J. van Wijk. Cvsscan: visualization of code evolution. In *SOFTVIS*, pages 47–56, 2005.

[147] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. J. Hendren, and C. Verbrugge. Evolve: An open extensible software visualisation framework. In *SOFTVIS*, pages 37–46, 2003.

[148] A. Whitaker, R. S. Cox, , and S. D. Gribble. Diagnosing computer problems using time travel. In *Proceedings of the 11th SIGOPS European Workshop*, Sept. 2004.

[149] M. Whitney, M. Bernstein, R. D. Mori, K. Kontogiannis, B. Corrie, H. M&#252;ller, S. Tilley, E. Merlo, J. Mylopoulos, K. Wong, J. H. Johnson, J. McDaniel, and M. Stanley. Using an integrated toolset for program understanding. In *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 59. IBM Press, 1995.

[150] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[151] P. Young and M. Munro. A new view of call graphs for visualising code structures, 1997.

[152] P. Young and M. Munro. Visualizing software in virtual reality. In *IWPC*, pages 19–27, 1998.

[153] M. Zelkowitz. Reverse execution. *Communications of the ACM*, 16(9), Sept. 1973.

F Biographical Sketches

F.1 Christian S. Collberg

Department of Computer Science, University of Arizona, Tucson, AZ 85721, (520) 621-6612, collberg@cs.arizona.edu

Professional Preparation

Exchange student	Tulane University, 1979-1980.
BSc	Computer Science and Numerical Analysis, May 1986, Lund University, Sweden.
Ph.D.	Computer Science, Dec. 1992, Lund University, Sweden. Advisor: Anders Edenbrandt.

Appointments

Jan. 1993 – Dec. 1993	Temporary lecturer, The University of Lund, Sweden.
Jan. 1994 – Dec. 1998	Lecturer (Assistant Prof. in the US system), The University of Auckland, New Zealand.
Aug. 1998	Continuation (similar to tenure in the US system) awarded, The University of Auckland, New Zealand.
Jan. 1999 – May 2006	Assistant Prof., The University of Arizona.
Jun 2006 –	Associate Prof., The University of Arizona.

Articles most relevant to this proposal

1. Christian Collberg, Automatic derivation of compiler machine descriptions, *ACM Transactions on Programming Languages and Systems*, Valume 24, Number 8, 2002.
2. C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, June 2004.
3. C. Collberg, S. G. Kobourov, S. Kobes, B. Smith, S. Trush, and G. Yee. Tetratetris: An application of multi-user touch-based human-computer interaction. *9th International Conference on Human-Computer Interaction (INTERACT)*, pages 81–88, 2003.
4. C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. *ACM Symposium on Software Visualization (SoftVis)*, pages 77–86, 2003.
5. Christian Collberg, Ginger Myles, Andrew Huntwork, The SANDMARK Software Protection Research Tool, *IEEE Magazine on Security and Privacy*.

Additional relevant articles

1. C. Collberg, E. Carter, S. Kobourov, and C. Thomborson. Error-correcting graphs. *Workshop on Graphs in Computer Science (WG'2003)*, June 2003.
2. Christian Collberg, Clark Thomborson and Douglas Low, Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs, *ACM Principles of Programming Languages* (POPL'98), San Diego, CA, Jan 1998.
3. Christian Collberg and Clark Thomborson, Software Watermarking — Models and Dynamic Embeddings, *ACM Principles of Programming Languages* (POPL'99), San Antonio, TX, Jan 1999.
4. Christian Collberg and Clark Thomborson, Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection, *IEEE Transactions on Software Engineering*, Volume 28, Number 8, pp. 735–746, August 2002.

5. Ginger Myles, Christian Collberg, *Software Watermarking Through Register Allocation: Implementation, Analysis, and Attacks*, ICISC 2003.

Synergistic activities

Service:

- Conference committees: ACM SIGPLAN PLDI'99, Compiler Construction 2002, Information Hiding 2004, Information Hiding 2005, Digital Rights Management: Technology, Issues, Challenges and Systems (DRMTICS 2005), Second Symposium on Intelligence and Security Informatics (ISI-2004).
- Journal reviews: IEEE Security & Privacy, Software Quality Journal, Crossroads special issue on Ethics and Computer Science, Software — Practice and Experience,Journal of Systems and Software.
- Conference reviews: ACM POPL, ACM PLDI, WG 2005, DYNAMO 1999, Electronic Commerce 2000, 20th Intl. Conf. on Distributed Computing Systems (ICDCS-20), ACM Symposium on Principles of Distributed Computing (PODC 2000), RSA 2002.
- Research proposal review: Innovation and Technology Commission for the Hong Kong Special Administrative Region, NSF (two panels), Dutch Technology Foundation.
- PhD reviewer, Queensland University of Technology.
- Community software: `SPlaT` is a tool for detecting self-plagiarism among academics (Collberg and Kobourov, Self-plagiarism in computer science, CACM, Volume 48, Number 4, 2005).

Teaching: Revamped graduate and undergraduate compiler and programming language classes. Educational software:

- `AlgoVista`: An Algorithmic Search Tool in an Educational Setting, *35th Technical Symposium on Computer Science Education (SIGCSE)*, 2004.

Software: Led the development of ALGOVISTA, SANDMARK, ART, and an automatic compiler retargeting tool. Participated in the development of `Slinky`.

Graduate and Post-Doctoral Advisors

Dr. Anders Edenbrandt (current affiliation unknown)

Thesis Adviser and Postgraduate-Scholar Sponsor

Edward Carter (Berkeley), Kelly Heffner (Harvard University), Douglas Low (University of Washington) Ginger Myles (IBM Almaden), Jasvir Nagra (University of Auckland), Ashok Ramasamy (current affiliation unknown), Michael Stepp (UC San Diego)

Total no. graduate students advised: 7

Total no. postdoctoral scholars advised: 0

Collaborators (last four years)

Clark Thomborson (University of Auckland), Todd Proebsting (Microsoft Research), Jasvir Nagra (University of Auckland), Stephen Kobourov, Steven Kobes, Ben Smith, Stephen Trush, Gary Yee, Edward Carter, Joshua Louie, Thomas Slattery, Suzanne Westbrook, Ginger Myles, Tapas Sahoo, Richard T. Snodgrass, Shilong Yao, Gregg Townsend, Kelly Heffner, John H. Hartman, Sridivya Babu, Sharath K. Udupa, Zachary Heidepriem, Armand Navabi, Michael Stepp, Saumya Debray, Cullen Linn (all at University of Arizona).

F.2 John H. Hartman

Department of Computer Science, University of Arizona, Tucson, AZ 85721.
Phone: 520-621-2733; Email: jhh@cs.arizona.edu

a. Professional Preparation

Sc.B.	Computer Science, Brown University, May 1987
M.S.	Computer Science, University of California at Berkeley, May 1990
Ph.D.	Computer Science, University of California at Berkeley, December 1994

b. Appointments

Associate Professor, University of Arizona, since 2001.
Research Fellow, Princeton University, Sept. 2003 to Jan. 2004.
Assistant Professor, University of Arizona, 1994–2001.

c.i Related Publications

[1] Christian Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa, "Slinky: Static Linking Reloaded". *Proceedings of the 2005 Usenix Technical Conference*, April 2005.

[2] Steve Muir, Larry Peterson, Marc Fiuczynski, Justin Cappos, and John Hartman, "Proper: Privileged Operations in a Virtualized System Environment" (short paper), *Proceedings of the 2005 Usenix Technical Conference*, April 2005.

[3] Larry Peterson, Yitzchak Gottlieb, Steve Schwab, Mike Hibler, Patrick Tullmann, Jay Lepreau, and John Hartman, "An OS Interface for Active Routers", *IEEE Journal on Selected Areas in Communications –Active and Programmable Networks*, 2001.

[4] John H. Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck, "Experiences Building a Communication-Oriented JavaOS", *Software: Practice & Experience* 30, 10, August 2000, 1107–1126.

[5] John H. Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck, "Joust: A Platform for Communications-Oriented Liquid Software", *IEEE Computer* 32, 4, April 1999, 50–56.

c.ii Additional relevant articles

[1] John H. Hartman, Scott Baker, and Ian Murdock, "Customizing the Swarm Storage System using Agents", *Software: Practice & Experience*. To appear.

[2] Scott Baker and John H. Hartman, "The Design and Implementation of the Gecko NFS Web proxy", *Software: Practice & Experience* 31, 7, (2001), 637–665.

[3] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson, "Optimizing TCP Forwarder Performance", *IEEE/ACM Transactions on Networking* 8, 2 (April 2000), 146–157.

[4] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson, "Toba: Java For Applications: A Way Ahead of Time (WAT) Compiler", *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, June 1997.

[5] John H. Hartman, Ian Murdock, and Tammo Spalink, "The Swarm Scalable Storage System", *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, June 1999, 74–81.

d. Synergistic activities

- **Service**
  - Program committees: Supercomputing ’02, OpenArch ’02, OSDI ’99, IOPADS ’99, SIGMETRICS ’98, and ICPP ’98.
  - NSF Panels: PDOS 2005, CAREER 2003
  - Local arrangements chair for HotOS-VII, April 1999.
- **Teaching**
  Revamped graduate and undergraduate operating systems course curricula. Helped redesign undergraduate program curriculum to streamline prerequisites. Helped develop undergraduate course on system software and assembly language programming.
- **Software Produced**
  Participated in the development the Sprite and Scout operating systems and the TopoVista GIS system. Led the development the Zebra, Gecko, Swarm, and Mirage file systems. Led the development of the Stork environment service, the Toba run-time system for Java, and the Joust system for active networks. All of these systems have been made publicly available.

e. Collaborators and other affiliations

(i) **Recent Collaborators** (within the last 48 months) :
Larry Peterson (Princeton University), Yitzchak Gottlieb (Princeton University), Steve Schwab (Network Associates), Mike Hibler (University of Utah) Patrick Tullmann (VMware) Jay Lepreau (University of Utah), Hrishikesh Dandekar, Andrew Purrell (current affiliation unknown), Prasenjit Sarkar (IBM Almaden), Andy Bavier (Princeton University), Peter Bigot (Rincon Research), Patrick Bridges (University of New Mexico), Brady Montz (current affiliation unknown), Rob Piltz (current affiliation unknown), Todd Proebsting (Microsoft Research), Oliver Spatscheck (AT&T Labs) Jorgen S. Hansen (current affiliation unknown), Tammo Spalink (Princeton University), Garth Gibson (CMU), Scott Baker (University of Arizona), Ian Murdock (Progeny) Steve Muir (Princeton University) Marc Fiuczynski (Princeton University) Justin Cappos (University of Arizona) Sridivya Babu (University of Arizona) Sharath Udupa (University of Arizona)

(ii) **Graduate and Post-Doctoral Advisers** :
John Ousterhout, Electric Cloud, Inc.

(iii) **Thesis Adviser and Postgraduate-Scholar Sponsor**:
Scott Baker (University of Arizona)
Justin Cappos (University of Arizona)
Wanda Chiu (current affiliation unknown)
Huilong Hui (University of Arizona)
Prasenjit Sarkar (IBM Almaden)
Tammo Spalink (Princeton University)

Total no. graduate students advised: 6

Total no. postdoctoral scholars advised: 0

F.3 Stephen G. Kobourov

Department of Computer Science, University of Arizona, Tucson, AZ 85721.
Phone: 520-621-4324; Email: kobourov@cs.arizona.edu

Education

DARTMOUTH COLLEGE	Computer Science, Mathematics	B.S. *(summa cum laude)*	1995
JOHNS HOPKINS UNIVERSITY	Computer Science	M.S.	1997
JOHNS HOPKINS UNIVERSITY	Computer Science	Ph.D.	2000

Appointments

Since 2000	Assistant Professor, Computer Science, UNIVERSITY OF ARIZONA
Summer 2004	Visiting Researcher, LANCASTER UNIVERSITY, LANCASTER, UK
Summer 2003	Visiting Researcher, DIMACS, RUTGERS UNIVERSITY
1998 – 1999	Visiting Instructor, Computer Science, DARTMOUTH COLLEGE

Five Publications Most Related to this Project

1. P. Gajer and S. G. Kobourov, "GRIP: Graph Drawing with Intelligent Placement," *Journal of Graph Algorithms and Applications*, vol. 6, no. 3, p. 203–224, 2002. (Invited to this special issue on the best papers from GD'2000.)
2. P. Gajer, M. T. Goodrich, and S. G. Kobourov, "A Multi-Dimensional Approach to Force-Directed Layouts of Large Graphs," *Computational Geometry: Theory and Applications*, vol. 29(1), p. 3-18, 2004. (Invited to this special issue on the best papers from CGC'2001.)
3. S. G. Kobourov and K. Wampler, "Non-Euclidean Spring Embedders," *IEEE Transactions on Visualization and Computer Graphics*, to appear in 2005. (Preliminary version appeared in *10th IEEE Symposium on Information Visualization (INFOVIS)*, p. 207–214, 2004.)
4. C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A System for Graph-Based Visualization of the Evolution of Software," *ACM Symposium on Software Visualization (SOFT-VIS)*, pp. 77–86, 2003.
5. B. Dux, A. Iyer, S. Debray, D. Forrester, and S. G. Kobourov, "Visualizing the Behavior of Dynamically Modifiable Code." *13th IEEE International Workshop on Program Comprehension (IWPC)*, p. 337-340, 2005.

Five Additional Most Significant Publications

1. C. A. Duncan, M. T. Goodrich, and S. G. Kobourov, "Balanced Aspect Ratio Trees: Combining the Advantages of k-d Trees and Octrees," *Journal of Algorithms*, vol. 38, p. 303–333, 2001. (Invited to this special issue on best papers from SODA'99.)
2. C. A. Duncan, M. T. Goodrich, and S. G. Kobourov, "Balanced Aspect Ratio Trees and Their Use for Drawing Large Graphs," *Journal of Graph Algorithms and Applications*, vol. 4, p. 19–46, 2000. (Invited to this special issue on best papers from GD'98.)
3. C. Erten and S. G. Kobourov, "Simultaneous Embedding of a Planar Graph and Its Dual on the Grid," *Theory of Computing Systems*, vol.38(3), p.313-327, 2005. (Invited to this special issue on the best papers from ISAAC 2002).
4. C. Collberg, S. G. Kobourov, S. Kobes, B. Smith, S. Trush, and G. Yee, "TetraTetris: An Application of Multi-User Touch-Based Interaction using DiamondTouch," *9th International Conference on Human-Computer Interaction (INTERACT)*, p. 81–88, 2003.
5. C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee, "GraphAEL: Graph Animations with Evolving Layouts," *11th Symposium on Graph Drawing (GD)*, p. 98–110, 2003.

Synergistic Activities

Conferences

- Steering Committee, Graph Drawing, 2001–current
- Editorial Board, Graph Drawing E-print Archive, 2005–current
- Organizer, Dagstuhl Workshop on Graph Drawing, Dagstuhl, Germany, 2005
- Chair of Program Committee of 10th Symposium on Graph Drawing (GD), 2002
- Program Committee, 17th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2006
- Organizing Committee, 22nd ACM Symposium on Computational Geometry, 2006
- Chair of 12th Annual Graph Drawing Contest, Limerick, Ireland, 2005
- Chair of 11th Annual Graph Drawing Contest, New York, NY, 2004

Journal Ref.

SIAM Journal of Computing, ACM Transactions on Graphics, Journal of Algorithms,Journal of Discrete and Computational Geometry, Journal of Discrete Applied Mathematics, Journal of Graph Algorithms and Applications, Journal of Computational Geometry: Theory and Applications, Journal of Combinatorics, Algorithmica, Networks, IEEE Transactions of Visualization and Computer Graphics, Software Practice and Experience

Conf. Ref.

Symposium on Graph Drawing (GD), ACM Symposium on Computational Geometry (SCG), ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM Symposium on Theory of Computing (STOC), IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Conference on Information Visualization (INFOVIS), European Symposium on Algorithms (ESA), Workshop on Algorithm Engineering and Experiments (ALENEX), Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)

Courses

Created new course at U Arizona: Information Visualization and Graph Drawing
Organizer of weekly seminar on Computational Geometry and Information Visualization

Educ. Tools

- "`SAIL`: Generating, Archiving and Retrieving Specialized Assignments In $\LaTeX$," *31st Technical Symposium on Computer Science Education, (SIGCSE)*, p. 300–304, 2000.
- "`PILOT`: An Interactive Tool for Learning and Grading," *31st Technical Symposium on Computer Science Education (SIGCSE)*, p. 139–143, 2000.
- "`AlgoVista`: An Algorithmic Search Tool in an Educational Setting," *35th Technical Symposium on Computer Science Education (SIGCSE)*, p. 462–466, 2004.
- "`AlgoVista`: A Tool to Enhance Algorithm Design and Understanding," *7th Ann. Symp. on Innovation and Technology in CS Education (ITiCSE)*, p.228–228, 2002.
- "Increasing Undergraduate Involvement in Computer Science Research," *8th World Conference on Computers in Education (WCCE)*, to appear in 2005.

Recent Collaborators (within the last 48 months)

J. Abello (DIMACS), T. Biedl (U Waterloo), F. Brandenburg (U Passau), P. Brass (CUNY), C. Collberg (U Arizona), S. Debray (U Arizona), E. Demaine (MIT), C. A. Duncan (U Miami), A. Efrat (U Arizona), D. Eppstein (UC–Irvine), R. Fleischer (Hong Kong U), H. H. Gonzáles-Baños (Honda Research), M. T. Goodrich (UC–Irvine), D. Ismailescu (Hofstra), V. S. A. Kumar (Los Alamos National Labs), G. Liotta (U Perugia), A. Lubiw (U Waterloo), J. S. B. Mitchell (Stony Brook), P. Mutzel (U Dortmund), C. Thomborson (U. Auckland), C. Wenk (U Texas–San Antonio), S. Westbrook (U Arizona)

Graduate and Post-Doctoral Advisors: Prof. Michael T. Goodrich (University of California)

Thesis Advisor and Postgraduate-Scholar Sponsor: Cesim Erten (ISIK University, Istanbul, Turkey)

Total no. of graduate students advised: 1; postdoctoral scholars sponsored: 0.

G Budget Justification

Senior Personnel. One month's summer support in 2005, 2006, and 2007 each for Hartman, Kobourov, and Collberg. Anticipated salary increases are 1.7% per year in years 2 and 3. Fringe benefit rate for faculty is 26.7%.

Fringe benefits have been computed as follows:

Year 1: 7 months @ 26.7%; 5 months @ 27.1%.

Year 2 and 3: 27.1%.

Graduate Students. Since the proposed research requires a significant amount of systems implementation, we request support for 2 doctoral students. They will be supported at 20 hrs/week during the academic year and 40 hrs/week during the summer. Anticipated stipend increases of 1.7% per year in years 2 and 3.

Fringe benefits have been computed as follows:

Year 1: 7 months @ 29.5% (18.3% is tuition remission that is exempt from indirect costs). 5 months @ 31.5% (20.9% is tuition remission that is exempt from indirect costs).

Year 2 and 3: 31.5% (20.9% is tuition remission that is exempt from indirect costs).

Postdoctoral Researcher. We request support for 1 post-doctoral researcher. This person (tentatively, Scott Baker who will be graduating from our department this year) will be leading the software development effort and help in advising the undergraduate students.

Undergraduate Students. We request support for 2 undergraduate-level students. The PIs have had much success mentoring undergraduates in the past. We get them involved with research early, and have been successful in getting them admitted into prestigious graduate programs based on their outstanding publication records. Students will be supported at 10 hrs/week during the academic year and 40 hrs/week during the summer.

Travel. We request support for one trip per PI and graduate student, for a total of six trips per year, to attend scientific conferences and present our results. We have budgeted approximately $1670 per trip, including air fare, hotel costs, and conference registration, for a total of $10,000 per year.

Capital Equipment. We request funds to purchase one Linux multiprocessor, three large displays optimized for graphics and text, and a workstation to drive the monitors. The following table breaks down the cost of a possible system:

Three 30" Apple Cinema displays, 2560x1600 pixels	3x$3000=$9,000
One Apple Dual G5 workstation to drive displays, 1GB RAM	$3,348
Two NVIDIA GeForce 6800 Ultra DDL graphics cards (each can drive two 30" displays)	2x$600=$1,200
Western Scientific FusionA8 8-way Opteron 848 (2.2GHz) server with 16GB RAM, 1.2TB disk, and four 1Gb NICs	$20,500
Rack, monitor, keyboard, mouse, cabling, display mounting hardware, etc.	$500
Total	$34558

We have chosen this particular display technology to allow us to view fast animations and high-resolution graphics *and* text. This will be necessary to adequately display source code along with our fly-through geographic graphics. At the present time, it is not possible to connect more than two 30" monitors to an Apple G5, but we expect solutions to appear during the course of the project.

Operations We request funds to purchase three desktop workstations, one for each PhD student and post-doctoral researcher supported by this grant, and four laptops for the undergraduate students:

Three desktop systems	3x$1,000=$3,000
Four laptops	2x$1,500=$3,000
Total	$6,000

This amounts to $6,000 in year 1. Additionally, we request $3,000 in years 2 and 3 for printer paper, toner cartridges, copying, phone calls, and similar items; page charges for journals; preparation and dissemination of documentation on the software systems that will be developed.

Indirect Costs. These are as follows:

Year 1: 7 months @ 50.5% of direct costs except for equipment; 5 months @ 51%.

Year 2 and 3: 51.0% of direct costs except for equipment.

H Current and Pending Support

Richard Snodgrass and Christian Collberg, *Tamperproof Audit Logs*, NSF IDM grant, September 1, 2005 - August 31, 2008, \$330,000. Christian Collberg is budgeted at \$2,500/summer.

I Facilities, Equipment, and Other Resources

The Computer Science Department is located on the 2nd, 7th, 8th, and 9th floors of the Gould-Simpson Science Building. We currently provide five computing laboratories within Gould-Simpson: A combined Graphics and Instructional Lab in Gould-Simpson 930 (a 42-station Pentium 4 based Windows XP and Linux PC facility with high resolution LCD monitors), a second Instructional Lab in Gould-Simpson 228 (a 56-station Pentium 4 based Windows XP and Linux lab), and three Research Labs in Gould-Simpson 748, 756, and 913. There are numerous short term project rooms scattered throughout our floors. The department maintains a three year (or sooner) replacement cycle on all computing equipment. Our newest computing addition is a 32-node Pentium 4 cluster supporting non-blocking, switched gigabit ethernet. We also have a 10-node Pentium 4 cluster on switched gigabit ethernet for computation-intensive projects. In an effort to harness wasted CPU power from otherwise idle desktop computers, the department offers CSGrid. Using the Wisconsin Condor software, workstations within the department are made available for remote batch computing. CSGrid is currently 24 nodes and growing.