Jeffrey Vetter
Lawrence Livermore National Laboratory

Karsten Schwan
Georgia Institute of Technology

Techniques for High-Performance Computational Steering

Computational steering lets researchers investigate, calibrate, and control long-running, resource-intensive applications at runtime. Magellan, a prototype computational steering system, uses ACSL to intelligently control multithreaded, asynchronous steering servers that cooperatively steer applications.

Simulations—both parallel and distributed—are playing an increasingly critical role in all areas of science and engineering. As the applications of these simulations expand, the demand grows for high-performance computing of increasing power, flexibility, and utility. Interactive computational steering is one way to increase the utility of high-performance simulations for scientists because it lets them drive the scientific discovery process and interact with their data. They can investigate, calibrate, and correct the execution of their simulations. For example, computational steering lets users interactively maneuver molecule docking in a protein-design simulation,[1] interactively examine and control structures' shape optimization,[2] and manage the process of simulated annealing.[3] Steering provides two major benefits: it lets users diagnose their simulation models, and it provides insights into the complex runtime interactions of the simulation components.

Computational steering is the online management of application execution and application resources to improve performance or explore an application (see the "Computational-steering issues and taxonomy" sidebar).[4] Initially, implementations of computational-steering systems focused on functionality with limited effort dedicated to understanding the performance of the steering-system architecture. Furthermore, application users customized many of these designs to specific operating environments and applications; therefore, these approaches were difficult to leverage across applications and target platforms. Nevertheless, interest in techniques for computational steering persists (see the "Related work" sidebar). Recent interest in virtual environments[4] has further fueled the need for an efficient, dynamic infrastructure for high-performance computational steering. Three reasons motivate the need for high performance in computational steering. The steering system must

- act intelligently at runtime to minimize its perturbation of the target application,
- analyze and forward to the user the data extracted from the target in a timely fashion to allow for up-to-date visualizations and decision making, and
- execute steering commands with low latency to prevent undesirable feedback.

We have developed methods for high-performance computational steering based on a domain-specific language called ACSL and on one runtime optimization example: spatial locality. Our language and software design provide a flexible, compatible, and efficient way to describe steer-

 1092-3063/99/$10.00 © 1999 IEEE

Computational-steering issues and taxonomy

Interactivity in high-performance computing is not a new topic. For years, scientists have interacted with their applications in various ways. Typically, users instrument their application software with the communications and graphics code required for runtime interactions. Such user interfaces are often quite effective because they are customized to particular application domains. However, because of such customizations, these interfaces are not easily leveraged across different applications. Moreover, application characteristics vary dramatically among applications as well as during separate phases of the same application. Therefore, the infrastructure should be dynamic. When either the user-issued steering request changes or the host system changes, the infrastructure must adapt. Additionally, visualization techniques implemented by a certain user interface simply might not be suitable for other applications, and these visualization and user interface issues for computational steering complement our research.[1]

Computational steering has two goals: application steering and performance steering. Application steering lets researchers investigate phenomena during the execution of their application, typically giving them structured access to various application objects. Users might change parameters, choose alternative algorithms, rollback and replay execution, or simply track application execution. Because of application steering's unpredictable nature, the steering system must provide access to many application objects, not knowing which objects to optimize a priori.

Performance steering, on the other hand, seeks to improve an application's performance by changing application and resource parameters during runtime. Manual load balancing is an example of interactive performance steering, where a user manually steers a data decomposition to improve performance. With performance steering, users have a general idea of which parameters influence performance. In fact, they might have determined which parameters to focus on by first using application steering or some other type of sensitivity analysis. In general, however, performance steering is often performed by algorithms,[2] because it can be quantified and automated. Algorithmic steering is also useful for long-running applications when human users are neither constantly available (for example, batch processing) nor capable of processing rapid changes to application data and parameters.

These two goals of application exploration and performance steering first appear mutually exclusive because each requires different results from the steering system itself. However, adaptable steering techniques can provide a common basis for various types of steering on different types of applications. Our language-based approach provides flexibility and abstraction that let the steering system interpret and optimize steering requests.

References

1. J.S. Vetter, "Computational Steering Annotated Bibliography," *SIGPLAN Notices*, Vol. 32, No. 6, June 1997, pp. 40–44.
2. R.L. Ribler et al., "Autopilot: Adaptive Control of Distributed Applications," *Proc. Seventh Int'l Symp. High-Performance Distributed Computing*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998, pp. 172–179.

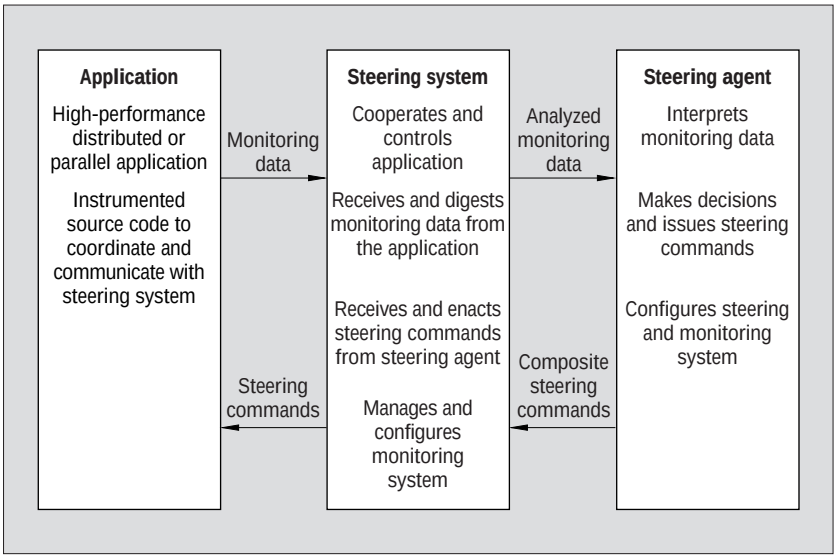

Figure 1. Abstract decomposition of a steering system's functionality. Monitoring data flows from the instrumented application through the steering system to the steering agent. The agent interprets the data and decides which steering commands to issue. Those commands flow back into the steering system. The steering system then makes changes to the executing application.

ing commands to the steering system. Our prototype computational-steering system, Magellan, can interpret and optimize these commands to meet various criteria, including application perturbation and high steering frequency—a new metric for steering systems.

System overview

We now define a more precise software architecture for any computational-steering system, which is essential for understanding the principles of our language approach. Figure 1 illustrates an abstract architecture for a computational-steering system. We assume that the user instrumented the application to coordinate with the steering system. At runtime, when the application generates monitoring information, it communicates this information to the attached steering system. The steering system coordinates and controls the application by receiving and digesting monitoring data from the

Related work

Both Steven Parker and his colleagues and Jeffrey Vetter provide thorough descriptions of research on computational steering and related fields.[1,2] Several systems for computational steering exist, each with its own specific contributions. Although computational steering has some similarities to other work in performance monitoring and debugging, we do not outline work in these areas.

The Visualization and Application Steering Environment first recognized the necessity of application control in addition to visualization.[3] VASE furnished application developers with tools to annotate their source code with steering breakpoints; it also provided an interface between itself and the application that did not depend entirely on a specific visualization or interaction method. D.J. Jablonowski and his colleagues developed VASE on a single-instruction, multiple-data platform, and this architecture's single instruction stream let VASE easily control the application through these steering breakpoints. VASE was one of the first prototypes to provide support for algorithmic steering through its scripting language.

Steven Parker and his colleagues demonstrated a programming environment for computational steering based on the dataflow architecture.[2] This environment essentially recasts computations as modules in a dataflow graph. Steering occurs when new data is injected into any module's input ports, and this, in turn, retriggers recalculation of dependent modules.

SMD interactively steers molecular dynamics calculations of protein molecules, including computation, visualization, and communication components.[4] Biochemists can "tug" molecules into different shapes by specifying external forces in the graphical interface, which are added to internal forces representing atomic bonds and nonbonded interactions.

Cumulvs provides computational steering and visualization for Parallel Virtual Machine applications.[5] The steering and visualization directives are integrated closely with the facilities of the PVM system to provide a transparent and portable tool.

Several studies have explored interesting problems in steering and related areas. For example, a study by Valerie Taylor and her colleagues explores lag time in an interactive virtual environment.[6]

We derived ACSL (our domain-specific language) and Magellan (our prototype computational steering system) from our earlier work on Progress and Falcon.[7,8] Progress and Falcon provide the ability to monitor and steer fine-grain items in an executing system. Falcon is an event-based monitoring system that provides low-latency access to application data at runtime. Progress is an application-specific, low-latency steering system that provides actuators and probes to enforce structured changes on executing systems.

References

1. J.S. Vetter, "Computational Steering Annotated Bibliography," *SIGPLAN Notices*, Vol. 32, No. 6, June 1997, pp. 40–44.
2. S.G. Parker, C.R. Johnson, and D. Beazley, "Computational Steering Software Systems and Strategies," *IEEE Computational Science & Eng.*, Vol. 4, No. 4, Oct.–Dec. 1997, pp. 50–59.
3. D.J. Jablonowski et al., "VASE: The Visualization and Application Steering Environment," *Proc. Supercomputing '93*, IEEE Computer Soc. Press, Los Alamitos, Calif., pp. 560–569.
4. J. Leech, J.F. Prins, and J. Hermans, "SMD: Visual Steering of Molecular Dynamics for Protein Design," *IEEE Computational Science & Eng.*, Vol. 3, No. 4, Winter 1996, pp. 38–45.
5. G.A. Geist II, J.A. Kohl, and P.M. Papadopoulos, "CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications," *Int'l J. Supercomputing Applications*, Vol. 11, No. 3, 1997, pp. 224–235.
6. V.E. Taylor et al., "Interactive Virtual Reality in Simulations: Exploring Lag Time," *IEEE Computational Science & Eng.*, Vol. 3, No. 4, Winter 1996, pp. 46–54.
7. J. Vetter and S. Schwan, "Progress: A Toolkit for Interactive Program Steering," *Proc. 24th Int'l Conf. Parallel Processing*, Vol. 2, CRC Press, Boca Raton, Fla., 1995, pp.139–142.
8. W. Gu et al., "Falcon: Online Monitoring and Steering Parallel Programs," *Concurrency: Practice & Experience*, Vol. 10, No. 9, 1998, pp. 699–736.

application, enacting steering commands issued from the steering agent, and managing and configuring the steering system itself. Human users represent steering agents and enter steering commands to the steering system. When the steering system receives these commands, it parses the specified command, possibly optimizing it. Then it enacts the command on the target application using precise, yet flexible, instrumentation mechanisms. Our criteria of efficiency, flexibility, and compatibility motivate our system design and language approach.

Efficiency

We recognize two separate efficiency goals. First, the steering agent controls application performance by controlling the degree of monitoring and steering perturbation on the application. If the agent decides not to monitor or steer an application for a particular episode, then the application runs normally. If, on the other hand, an agent wants to pause an application, investigate, and arbitrarily visualize data, then the application's performance should not be an issue. Second, the steering system performance itself remains important because high latency within the steering system can degrade decision quality and steering effectiveness.

Flexibility and compatibility

The steering system should work with various applications and steering agents in their existing environments. High-performance computing applications vary dramatically in both levels of concurrency and composition. The steering system's design should work with a range of these applications. Furthermore, the system should be dynamic enough to accommodate the varying requirements of visualization systems, ranging from Motif visualizations to remote, collaborative, immersive environments.

Magellan's software architecture

We have designed and implemented Magellan to investigate our approach's viability. Our system design provides a fundamentally different framework for

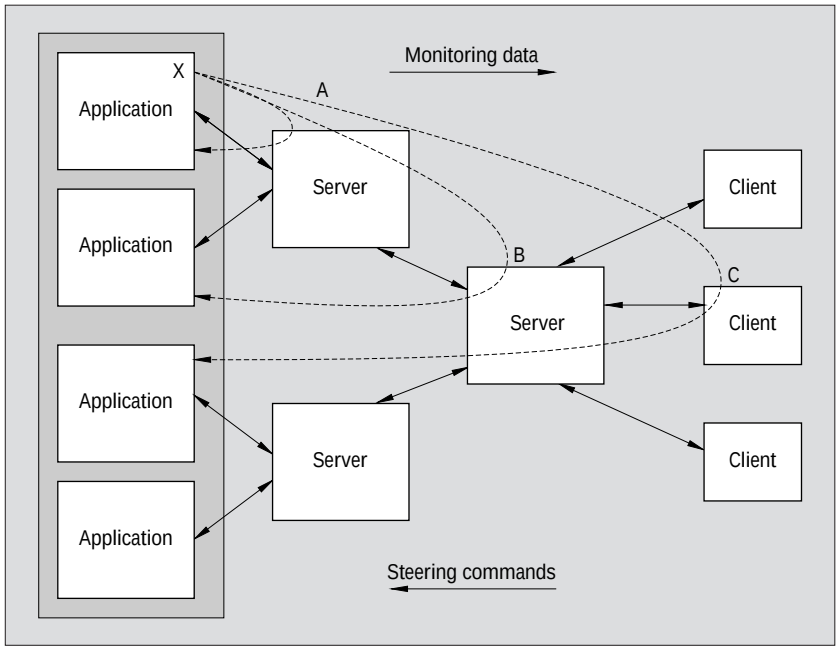

Figure 2. Magellan's software architecture.

interactive computational steering than that of previous systems. Figure 2 illustrates Magellan's basic architecture, which is a group of steering servers and clients. The application remains intact except for source-code annotations. These annotations identify application objects and how the server can change these objects in structured ways.

Steering servers represent the Magellan system's core. A steerable application always has at least one steering server connected to it. These servers are modular, and their number depends on the size and structure of the target application. However, the servers are lightweight, executing as threads in the application's address space. Steering servers communicate with steering clients, which we assume are transient. These clients, in turn, provide information to the steering agents and then receive commands from the agents; agents do not directly interact with steering servers. In Magellan, each level in the hierarchy, not only the feedback loop through the steering client and agent, is capable of feedback. ACSL provides the communication protocol between servers and clients, while servers control applications directly by accessing the executable. In Figure 2, the system components have higher throughput and lower latency moving to the left and more global knowledge and centralized control moving to the right.

Typically, shared-memory, thread-based applications have only one server. Distributed-memory applications, such as those using the message-passing interface, must have one server in every MPI process. These multiple servers connect to one master server, which is usually the server for processor 0. The other servers receive commands from and send data to this master server. The master server, then, communicates directly with the steering client.

Key design choices for the Magellan architecture

The primary design decisions essential to our approach include the need for application instrumentation, client-server architecture, and the use of an interpreter and language.

Application-specific instrumentation

Computational steering needs more information than that provided by techniques for debugging, monitoring, or executable editing. Normally, these other techniques require information about data and control flow that are generally found within the object files. However, Magellan needs to know not only which data to access but also how and when it can safely alter the data. Object files do not contain these last two items. Executable editing and debuggers rely on information stored by the compiler in object files. Using this information, executable editing lets users either statically or dynamically change the instruction sequences in object files. On the other hand, debuggers provide a wealth of information about the application's state. However, debuggers do not give users information about the semantic meaning of an instruction sequence. Hence, there is no information to assure users that their changes do not invalidate some ongoing computation. Automated tools cannot derive this semantic meaning solely from the application's source code or the executable.

Magellan solves these problems by providing source-code annotations for both data and control flow. Fortunately, using these annotations does not preclude using other techniques such as profiling or executable editing.

Client–server architecture

Magellan's software architecture is fundamentally a client–server design because it promotes three separate goals. First, steering servers are modular, and users can configure them to accommodate a range of application structures, including shared memory, message passing, and hybrids (for example, clusters). Second, servers can operate as a thread in the application's address space, which improves compatibility and efficiency. Third, this clean separation of clients from servers lets steering clients control the steering system from various sources, including remote, heterogeneous sources such as collaborative, immersive environments.

Interpreter and language

The domain-specific microlanguage and the runtime interpreter are fundamental to our approach to computational steering. The runtime interpreter dynamically optimizes requests described with a language while leveraging existing techniques for language optimization. Our language approach is congruent with visualization approaches to computational steering, because a graphical user interface can simply generate language statements to describe their steering requests, and the visualizations can bind visual objects to incoming language updates.

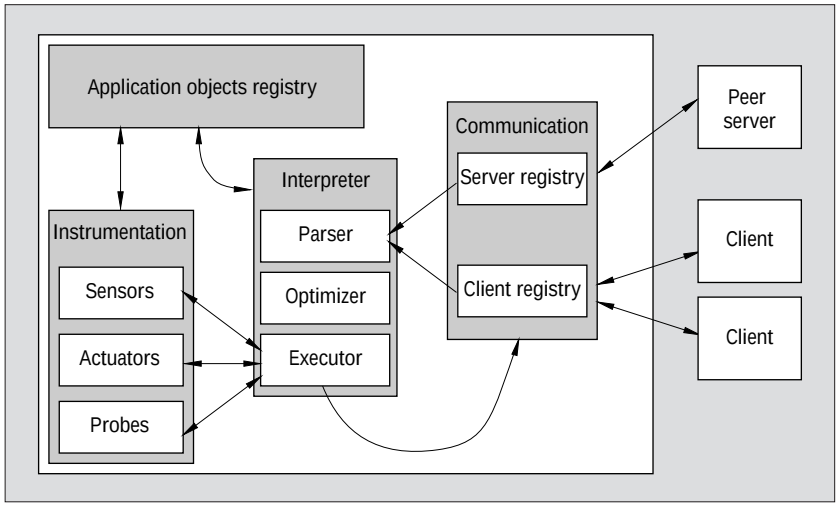

Figure 3. The internal architecture of one Magellan server.

Magellan server

As Figure 3 illustrates, the basic components of the steering server—the core of the Magellan system—are the interpreter with its optimizer, the application object registry, the client registry, the server registry, and controls for the monitoring and steering mechanisms. (This definition excludes graphics and user interfaces, which are traditionally provided by Magellan clients.) Most importantly, the interpreter receives and digests ACSL commands from clients and other servers. Then, it parses and optimizes these requests. Finally, it controls instrumentation to monitor and steer the application. The server contains three registries for all application objects, attached clients, and other servers.

Server design

This server design meets three goals. First, it gives the server fast access to any application object because it operates in the application's address space. Moreover, it gives the server access to the application's entire memory space without placing additional constraints on the application. For example, it does not force the developer to store steerable data in a Unix `mmap` region; although convenient, such a practice would prohibit access to application objects not explicitly stored in this shared-memory segment and most likely would degrade performance. Second, a modular server configuration easily reflects the design of target applications, such as shared memory, message passing, or a hybrid (clusters). This design lets servers operate alone, as peers, or as part of a hierarchy. This enhances scalability because servers can operate in a hierarchy with one master server. When the application uses distributed-memory models, such as MPI, our servers are distributed, because each server must operate as a thread in the application's address space. Alternatively, a shared-memory, threaded application has at least one server, but can have more. Finally, by encapsulating major steering functionality into the server, this design can run and maintain graphical interfaces and other visualization systems remote from the executing application.

Within the steering server, the application object registry contains information about all objects available in the application for steering, which the application declares to the server at runtime. The server needs this registry to access application objects. Object information includes the object's address, data type, size, and flags. The client registry maintains connection and caching information on Magellan clients; the server registry contains connection and management information on additional servers (if there are any).

Magellan interpreter

Magellan servers are multithreaded, asynchronous interpreters, and they provide two new capabilities for steering. First, the interpreter understands an LALR(1) (left-to-right-scan with one look-ahead token) grammar, which lets it accept user-constructed steering commands at any point during execution. Second, it can optimize these commands using various heuristics. In our optimization example, Magellan analyzes steering commands for spatial locality so that the server can migrate commands to the most efficient server.

Our decision to create a new language was not easy, and our contribution is not a new language, but an approach to computational steering that promotes high performance in the application as well as in the steering system. Because our language has a relatively small number of tokens and expressions, it performs well. Moreover, the interpreter is thread-safe so that multiple interpreters can execute concurrently in the same address space. However, ACSL is not as flexible as a general programming language and does not support user-defined extensibility.

When the interpreter receives an ACSL command, it parses the command into tokens and builds a syntax tree. The server can then optimize the command by annotating or rearranging the tree. Next, the server uses the application instrumentation to make changes to the application.

During the optimization phase, the interpreter analyzes the abstract syntax tree for the current command and may rearrange the tree slightly or annotate nodes in the tree with additional information. In case of spatial locality, the interpreter scans the syntax tree to determine the location of each object involved in the command. If the all objects and data in a statement can be executed locally by one server, then the steering system executes that expression at that server. If, on the other hand, the server cannot forward the command, then the server must execute the command locally and incur any necessary latencies.

By contrast, if the server did not have this optimization phase, the steering system would ship all the necessary monitoring data to the client, which would conduct the analysis. Unfortunately, during this data transmission, the application might have to be stalled, depending on the type of steering request. After the decision, the steering commands would be sent from the client back to the server. The Magellan spatial-locality optimization step avoids this long cycle, provided the optimization applies to a particular steering command.

```
 1 > "numProcs" probe read;          # probe read
 2   "numProcs" = {1 -1 {8}};
 3 > "tau" sensor enable;            # sensor enable
 4   "tau" = {1 8 {3.5460e-04}};
 5   "tau" = {1 9 {3.5780e-04}};
 6   "tau" = {1 10 {3.5813e-04}};
 7   "tau" = {1 11 {3.5891e-04}};
 8 > "tau" sensor disable;           # sensor disable
 9 > "tau" actuate {4.0e-4};         # actuate
10   "tau" = {1 -1 {4.0000e-04}};
```

Figure 4. Interaction with a runtime steering system using ACSL through a text-based steering client. The target application is the multigrid simulation, which continues to execute as the user enters steering commands through the steering client. User commands are indicated with a > prefix.

A domain-specific steering language

ACSL is a microlanguage for steering. For our steering examples, ACSL provided sufficient flexibility for different types of computational steering on our target applications. Users can create additional forms of steering by combining these primitives into expressions and statement lists. At a higher level of abstraction, expressions and statement lists provide Magellan with the majority of its optimization opportunities.

Using language specifications to address configuration and adaptation problems is not novel. In comparison to such work, our research focuses on understanding and evaluating the tradeoffs in application perturbation and steering latency resulting from alternative manners where the runtime system processes steering commands into online steering actions.

Primitives

ACSL commands specify control of monitoring and steering mechanisms to the server. It assumes that an application exports monitoring and steering objects using registration instrumentation.[5] These objects are not related to object-oriented languages and do not provide inheritance or polymorphism. They are simply application components that are named. Applications register these objects at runtime; they can be almost any registered application component, including integers, floats, doubles, and vectors of such types. Once registered, these objects are available for the steering system to inspect and manipulate in structured ways, with various instrumentation mechanisms.

ACSL provides structured operations that use registered objects as the target for their monitoring and steering expressions. ACSL is specified as an LALR(1) grammar with expressions using prefix notation. A typical ACSL steering command has the form `"<object name>" <action> [options] [data|expr]`. The `<object name>` is the name of an application object that is the command's target. The ACSL primitive `<action>` is described later. Actions usually require either `data` or `expr`. data is a constant while `expr` is a basic mathematical operation employing the target object itself or some object within the scope of the server's object registry. That is, objects within a server's scope are registered with that server or a child server.

Probes

Probes are the simplest instrumentation mechanism, much simpler than sensors and actuators. A probe read simply retrieves a value for an object from the application, regardless of the application's current state or calculation. Probe writes, which are the opposite of probe reads, overwrite a particular data item in application memory. ACSL provides `"<object name>" probe <read|write <data|expr>>` to probe read or probe write a particular object. In contrast to sensors and actuators, probes do not have any safeguards against reading or writing a value that the application is using. In Figure 4, the user probe reads `numProcs` in line 1. The Magellan server reads the value and returns a tuple that includes `numProcs`' current value: 8.

Sensors

Sensors are monitoring mechanisms that provide consistent information about application execution in the form of events. ACSL provides `"<object name>" sensor <enable|disable|freq <rate>>` to actually control sensors for a particular object. In line 3 of Figure 4, the user enables a sensor for `tau`, and sensor events begin arriving for the `tau` application object. In the application, `tau` is a floating-point number. The number of events arriving at the client after the user enables a sensor depends on the number of instrumentation points in the application and on the frequency in which the application threads encounter these instrumentation points. Lines 4 through 7 show the sensor events for `tau`. Line 8 disables the sensor for `tau`.

Actuators

An actuator assigns a new value to an object, as specified by the steering agent with data or an expression.[5] ACSL provides `"<object name>" actuate <data|expr>` for actuators. Unlike probes, when the application actuator accepts the command, it assures the steering system that the object data is consistent and available for modification without fear of corrupting application execution. In Figure 4, line 9 uses an actuator to request that the value of `tau` be changed to 4.0e–4 when the application has a safe opportunity to change it. (This safe opportunity is a developer-inserted source code annotation.) The `actuate` call returns the new value for `tau` when the actuator physically changes the value in the application.

Control structures

Using control structures, any user can compose into statement lists the ACSL commands that manipulate these sensors, probes, and actuators (see Figure 5). These control structures are based on the event-action paradigm, where some code block is executed when a monitoring event satisfies a predicate. ACSL

provides `when <expr> do {code block}` as the basic event-action control structure. Each `expr` can be a combination of objects and basic numerical expressions, as well as relational operators. Multiple objects may appear in each expression.

The `code block` is a list of steering commands that may be parameterized so that objects in the `code block` are available for use in its calculation. Parameterized objects are those steering objects referenced in the expression's conditional. As the parser consumes the expression, it assigns a number to each of these objects. The parser assigns numbers from left to right, beginning with one at the first object and incrementing the counter for each new object. The server resets the counter for each steering expression when parsing begins. The remaining portion of the expression then references these objects with the format `$n`, where `n` is the number that the object has been assigned during the parse.

Complex commands that involve event-action control structures require more analysis by the server. Consider the spatial-locality optimization again. When the server receives a statement list, it parses the list to determine if the objects in the expression and code block are local to one server. If the list satisfies this requirement, the server can enact the steering command (`code block`) locally to begin executing it without sending the monitoring data to the client. This improves both the monitoring data's and the steering enactment's latency.

Next, the server stores a parsed expression in its event-action table with a unique identifier. In the example, the expression `< "thread.*.load" 0.8` receives an identifier, and each object contained in the `expr` stores this identifier. Whenever an object's data is updated, `<expr>` is reevaluated because it has a pointer to this expression via the identifier. The event-action table entry contains a pointer to the `code block` for the control structure. When `expr` is true, the corresponding `code block` is executed. The overhead for evaluating this expression is a direct function of the expression complexity.

```
# Event Expression
when (< "thread.*.load" 0.8) do
{
  #Grow columns by 10% of problem width
  "$1.startColumn" actuate (- "$1.startColumn"
    (* "dimLength" 0.05));
  "$1.endColumn" actuate (+ "1.endColumn"
    (* "dimLength" 0.05));
}
```

Figure 5. Sample steering command in ACSL that uses the `when` control structure to specify changes in the column allocation to each thread. Actuator instrumentation points in the application safeguard application state by verifying requested changes and by using preconditions and postconditions.

Application instrumentation

Magellan instrumentation provides well-defined mechanisms to modify an executing application. Magellan servers monitor and steer an application using three basic mechanisms: sensors, actuators, and probes.[6] In line with our earlier high-performance design criteria, we present microbenchmarks for each of our basic low-level monitoring and steering mechanisms, which must be efficient. The following microbenchmarks detail the time consumed by the application thread when executing an instrumentation point for one of these mechanisms. This microbenchmarking helps with later analysis of the entire system. Although perturbation is an issue for all performance-monitoring systems, our objective here is to convince users that they will not suffer extensive performance degradation due to instrumentation.

An SGI Origin 2000 provided the target platform for our performance tests. The system has 32 MIPS R10000 microprocessors with a clock speed of 195 MHz (1/195 MHz ≈ 5.1 ns). Peak performance per processor is 390 Mflops. Data and instruction caches are 32 Kbytes, and the secondary unified cache is 1 Mbyte. Total memory is 8 Gbytes. The system uses the Irix 6.4 operating system. For reference, we measured a null procedure call at 68 ns.

Sensors

Sensors are instrumentation points in the application that capture a component of application state. When any application thread encounters this instrumentation point, it captures one snapshot of the target application state and forwards it to the steering server, usually through shared memory.[5,6] Figure 6 illustrates the pseudocode for a tracing sensor, and Figure 7 shows an instrumentation point in an application. A sensor fires when an application thread encounters an enabled sensor instrumentation point. When the instrumentation point fires, the sensor creates an event by copying the class, current time, thread identifier, and application state into the event. The

```
trace-sensor ( datahandle )
  hits++
  if enabled
    timestamp
    captures++
    allocate event
    copy datahandle to event
    package event
    while buffer_insert fails
      buffer_insert event
  return
```

Figure 6. Tracing sensor pseudocode.

```
 1 void multig (my_id)
 2 /*   perform multigrid (w cycles)              */
 3 int my_id;
 4 {
 5    /* declarations and initialization...       */
 6    AS_RegisterObject(steer, "residual norm",
 7     &multi->err_multi, AST_DOUBLE, 0, my_id,
 8     &sResNorm);
 9    /*...compute with multi->err_multi...       */
10   AS_Sense(sResNorm);
11   /*...compute with multi->err_multi...        */
12 }
```

Figure 7. One sensor instrumentation point in source code. `AS_RegisterObject` notifies the Magellan server that the object `"residual norm"` is available for monitoring. Then, the `AS_Sense` function uses the object handle to capture the state of `"residual norm"` and then forward it to the Magellan server, if the sensor is enabled.

sensor, then, places the event into a shared-memory first-in, first-out buffer. The application sensors produce events, and steering servers consume them. The Magellan server polls these FIFO buffers and extracts these tuples from the buffer for analysis. If a FIFO buffer is full or locked by the server, then the sensor instrumentation point must stall and retry inserting the event into the buffer.

For sensors, two perturbation costs are important: execution time for disabled sensors and execution time for a firing sensor. First, there should be little overhead experienced for disabled sensors. Applications will most likely be instrumented with far more sensors than actually used at runtime, allowing for online selection in application steering. The average cost for a disabled sensor is 190 ns, which is about 2.79 times the cost of a null procedure call. Second, the cost of a firing sensor must be small so that using a sensor is not prohibitively expensive. An enabled sensor that transfers 4 bytes of application state executes in 23 microseconds, which is about 121 times the cost of the disabled sensor. The increase in cost over the disabled sensor is due to three primary items:

- a mutex lock guarding the FIFO between the application and the steering server,
- allocation of memory for copying state, and
- copying of the object state to the monitoring event.

These results are reasonable, but more aggressive strategies (such as optimizing memory management) could improve an enabled sensor's execution time. Alternatively, Magellan sensors can also change their sampling rate to amortize costs across many sensor firings.

Actuators

Actuators are instrumentation points in an application (see Figure 8).[6] Actuators modify the state of an application component. Servers control actuators by sending requests to the actuator through a shared FIFO buffer, much like sensors send events to the server. When any application thread encounters this instrumentation point, it verifies any preconditions or postconditions; then it updates the target application component with this new application state. In this case (unlike sensors), the steering servers generate change requests, and actuators in the application consume these requests. Similar to sensors, servers use a shared-memory FIFO buffer to send requests to actuators. An actuator fires when two conditions occur: the steering server arms the actuator by placing an event in the FIFO buffer, and an application thread encounters an actuator instrumentation point, removes the event from the FIFO buffer, and makes the change to the application component. Otherwise, an actuator is dormant.

As with sensors, applications will most likely have many more actuators than are actually used at runtime. Therefore, a dormant actuator's perturbation costs must be minimized. The disabled actuator costs 137 ns, twice the cost of a procedure call. The enabled actuator is similar in execution time to the enabled sensor, costing about 25 μs; it costs only 182 times the disabled actuator when the data size is 4 bytes. The instrumentation mechanism for the actuator suffers the same implementation constraints as the sensor does: a mutex lock and two memory copies. Varying the actuator data size has effects similar to varying the data size for sensors because they both must acquire a mutex lock, copy memory, and either allocate or free memory.

Probe reads and probe writes

Probes are simpler than sensors and actuators. These mechanisms require no instrumentation points in the application. The Magellan server, because it occupies the same address space as the application, simply reads and writes `application state` directly. The server does not synchronize with the application, which can lead to inconsistencies in the recorded or changed data; in fact, these possible inconsistencies motivate the need for sensors and actuators. Probes' execution costs do not contribute to perturbation other than secondary effects such as cache and processor execution disruptions.

Evaluation

To evaluate our steering system, we (1) instrumented the application source code and recompiled the application, (2) ran the application with the steering system enabled, (3) steered the system (with an algorithm for performance steering), and (4) compared performance results.

Several design criteria are not easily quantified. For example, the ease of instrumentation is quite subjective. Therefore, we focus on quantitative aspects of the system. As we mentioned earlier, performance has many dimen-

sions for computational steering. Accordingly, our evaluation compares ACSL's performance with that of other steering strategies.

We chose to focus on performance steering with algorithmic clients to create consistent and repeatable performance results. Note that application steering is accomplished in a similar manner, but is more difficult to quantify. For these performance tests, the steering clients were simple C programs that controlled the steering servers.

```
bool verify (void *p)
{
    /* is *p valid? */
}

double tolerance = DEFAULT_E;
void main(argc, argv)
int argc;
char *argv[];
{
  /* declarations and initialization...        */
  AS_RegisterObject (steer, "tolerance",
   &tolerance, AST_DOUBLE, 0,
   &objHandle);
  /*...compute with tolerance...               */
  AS_Actuate( objHandle, verify, pre, post);
  /*...compute with tolerance...               */
}
```

Figure 8. One actuator instrumentation point in source code. `AS_RegisterObject` notifies the Magellan server that the object `"tolerance"` (a global variable) is available for steering. Then, if requested by the Magellan server, the `AS_Actuate` function uses the object handle to change `"tolerance"`. The user-defined function `verify` receives a pointer to the incoming data, and returns a Boolean value dictating whether the requested change is valid. The user-defined functions `pre` and `post` allow user code to be executed before and after the change.

Example applications

We use two different applications for our performance measurements: a 27-point 3D-stencil code and a multigrid simulation.

27-point 3D-stencil simulation

The heat-diffusion simulation is a 27-point 3D-stencil time-stepped simulation, which uses threads with a single-process, multiple data model. The data domain is decomposed into slabs along the z-axis. Each thread has two boundaries identifying its slab: a start and an end column. These columns are integers that are shared between neighbor threads and protected by mutex locks. Moving these boundaries changes the slab decomposition and balances simulation load. Experiments with this example used 128^3 cells in the 3D space, with heat as the cell quantity.

Multigrid simulation

For a more complex example, we use the Splash 2 benchmark, Ocean. We chose this application because it exhibits characteristics similar to various high-performance computing applications. Ocean simulates eddy and boundary currents in a cuboidal ocean basin. The application uses finite-differencing computational fluid dynamics with a regular grid. The key algorithm uses a red-black Gauss-Seidel multigrid equation solver; each time step in the simulation involves setting up and solving a set of spatial partial differential equations.

Application performance

To determine how our instrumentation and steering infrastructure affect application performance, we measured application performance on both the multigrid and the heat-diffusion simulations under different instrumentation scenarios. All times are wall-clock times. Table 1 gives the application-performance measurement results. Optimized mode is the uninstrumented, optimized application's original runtime. Heat diffusion runs on two processors for 204 seconds, with work evenly distributed between processors. Multigrid runs on two processors for 132 seconds. Instrumented mode is the application's runtime with all the necessary instrumentation for steering, but we performed no steering. That is, the steering system captured no monitoring events and executed no steering commands.

In these tests, one steering server executed concurrently with each application during each test. Both heat diffusion and multigrid ran within 10% of their original runtimes, with instrumentation at 223 and 137 seconds, respectively. This result bolsters the argument that applications can be instrumented for interactivity and still provide adequate performance. As a benchmark, debug mode demonstrates the runtimes of these unoptimized, uninstrumented applications with debugging information. In these examples, the unoptimized debug mode extends the runtime by more than 100% over that of the optimized mode. The perturbation induced on an application during steering is a function of the steering requests issued by the client.

Alternate design strategies for steering systems

Steering-system designers have a choice of several strategies. The steering-system infrastructure's design can dramatically affect its performance and, in fact, determine the types of steering possible. The primary difference in the following strategies is the range of possible latencies in the closed-loop steering sys-

tem. Only strategy 3 lets the steering system exploit the shorter feedback loop, because steering requests are encoded with ACSL, and the interpreter can make intelligent decisions about how to fulfill steering requests.

Strategy 1: brute force

The steering system forwards to the steering agent all raw monitoring data generated by the application. Once the agent interprets this data, it decides on steering commands, if any. The agent, then, issues these commands through the steering system. The advantage of this scenario is that it is relatively easy to implement. However, a major disadvantage is that it forces a relatively long feedback loop on all steering commands. Unfortunately, data congestion and management of the application can prohibit high performance. Furthermore, latency due to the long feedback loop can cause poor decision quality in the steering system.

Strategy 2: monitoring data analysis and steering-command synthesis

The steering system digests and filters raw monitoring data, which is then forwarded to the steering agent. The steering agent issues steering commands as compound commands. Then, the steering system coordinates and controls the execution of the commands. This strategy combines monitoring data analysis with steering-command synthesis. An advantage is that the system can filter data early in the process to reduce data congestion. The steering system can coordinate and expand these commands. On the other hand, although monitoring data latency might be reduced because of data filtering, any transmission latency in the steering system is forced on all this monitoring data. Steering commands return via the same path. In summary, all mandatory communication latencies remain similar to the brute-force strategy.

Strategy 3: Magellan steering language

Our language-based approach lets the steering client describe the steering commands and any conditions with a language. Because a language explicitly defines relationships between monitoring data and steering commands, the steering system can optimize the agent's request. Such a steering system can integrate monitoring and steering, then optimize the commands for spatial locality. In contrast to the aforementioned strategies, this choice has an advantage in that it allows dramatic changes in the operation of the steering system for two basic reasons:

- Massive amounts of monitoring data might never have to be transferred across the steering system to the steering agent, because these decisions are made locally.
- The frequency of steering commands increases dramatically, because these local decisions circumvent mandatory communication latencies.

This approach's major limitation is that the steering system can optimize only those language requests that fulfill its prerequisites. In some cases, the steering system simply degrades to strategy 2.

Table 1. The steering system's effects on application performance.

Application	State	Average runtime (seconds)	Relative performance
Multigrid	Optimized	132	1
	Instrumented	137	1.04
	Debug	312	2.37
Heat diffusion	Optimized	204	1
	Instrumented	223	1.09
	Debug	500	2.45

Steering frequency

To compare these alternate strategies, we developed a new steering metric called steering frequency. Equation 1 describes system latency.[7] The cost for one steering cycle in the steering feedback model includes generating monitoring data in the application Δ_{me}, transporting it to the steering agent Δ_{mt}, digesting the data Δ_{ma}, deciding on a steering command Δ_{agent}, issuing the steering command Δ_{ss}, transporting it to the steering system Δ_{st}, and executing the steering command Δ_{se}.

$$\begin{aligned}\Delta_{steercycle} &= \Delta_{monitor} + \Delta_{agent} + \Delta_{steer} \\ &= (\Delta_{me} + \Delta_{mt} + \Delta_{ma}) + \Delta_{agent} + (\Delta_{ss} + \Delta_{st} + \Delta_{se})\end{aligned} \quad (1)$$

Over a period of one second, the number of possible steering events per server based on the most recent monitored data is limited to $\omega_{steering}$, as specified in Equation 2. Steering frequency (Equation 2) is a function of latencies that exist throughout the closed-loop steering system. We use this metric to compare the brute-force strategy (which is typical of many steering systems) with the steering-language approach.

$$\omega_{steering} = \frac{1}{\Delta_{steercycle}} = \frac{1}{\Delta_{monitor} + \Delta_{agent} + \Delta_{steer}} \quad (2)$$

Increasing any of these components decreases the steering frequency. Consequently, a higher frequency indicates that the entire system is operating efficiently, because these latencies are low.

Steering frequency for brute-force steering system

Most of the different components of a brute-force steering system have been studied independently. However, the entire steering cycle depends on the combination of these components and the particular implementation. Gathering data with a monitoring system such as Magellan, we can expect a steering system to have a Δ_{me} of about 23 μs. Furthermore, a steer-

ing system such as Magellan can arm actuators in a similar amount of time. Ignoring the other costs, $\omega_{steering}$ is 1/(23 + 23) μs ≈ 21,739 Hz.

Transport times for both monitoring and steering depend greatly on the underlying network protocol and operating system. TCP/IP, for example, has a single one-way message send of about 1 ms over a traditional 10-Mbyte Ethernet on Unix. Adding the transportation costs to $\omega_{steering}$ produces 1/(46 + 1,000 + 1,000) μs ≈ 488 Hz.

The response time for the steering agent is directly a function of the complexity of the steering agent. The agent can be as simple as a five-line C program or as complex as a virtual reality environment. Assuming there is a human in the loop for interactivity, we can ignore the actual costs of the interface technology and conservatively estimate the human user response time of about 100 ms. Given this estimate, the final $\omega_{steering}$ is 1/(46 + 2,000 + 100,000) μs ≈ 10 Hz. The costs for this architecture are $\Delta_{agent} >> \Delta_{mt}$, $\Delta_{st} >> \Delta_{me}$, Δ_{ma}, Δ_{ss}, Δ_{se}, which leads to hierarchical costs and limitations for this type of steering system. This conservative estimate demonstrates that the frequency of interactive computational steering with a brute-force system approaches 10 Hz. By forcing all monitoring events and steering commands to flow through the entire loop, this strategy limits the frequency with which the system can respond to applications and users.[7] Algorithmic steering would certainly improve this degradation by reducing the agent response time, but this would also limit steering flexibility.

Steering frequency for the Magellan steering system

The language strategy (strategy 3) changes Equation 2 to recognize that ACSL commands can take advantage of spatial locality and obviate mandatory communication latencies. If the optimizer determines that the requested command can be propagated closer to the application with respect to latency, it does so. This opportunity vastly improves steering frequency by reducing the latencies of steering commands. First, because the steering command is close to the application, the steering system can receive monitoring information at higher rates and execute steering decisions at a higher frequency. Second, because the server digests monitoring information early in the process, the steering infrastructure is not overwhelmed with data. Third, ACSL maintains flexibility because the system migrates steering commands to the servers only when the commands satisfy the optimization prerequisites. Furthermore, because users can compose ACSL commands, this strategy does not limit the types of steering for the application.

For Magellan, on the heat-diffusion code, $\Delta_{steercycle}$ was 49 μs for steering performed directly at the steering server, which led to a $\omega_{steering}$ of approximately 20,400 Hz. Forcing Magellan to route data through the client raised $\Delta_{steercycle}$ to approximately 3 ms, dropping the steering frequency to 333 Hz.

The performance steering results in Figure 9 demonstrate the importance of steering frequency. In this experiment, we changed only the steering action's latency. We executed the heat-diffusion application, with Magellan balancing the load as described earlier. In this configuration, we tested two different steering frequencies for the application. An efficient steering system, such as Magellan with its language approach, can have a high steering frequency of 20,000 Hz, because commands are closer to the application and small amounts of data must cross process boundaries.

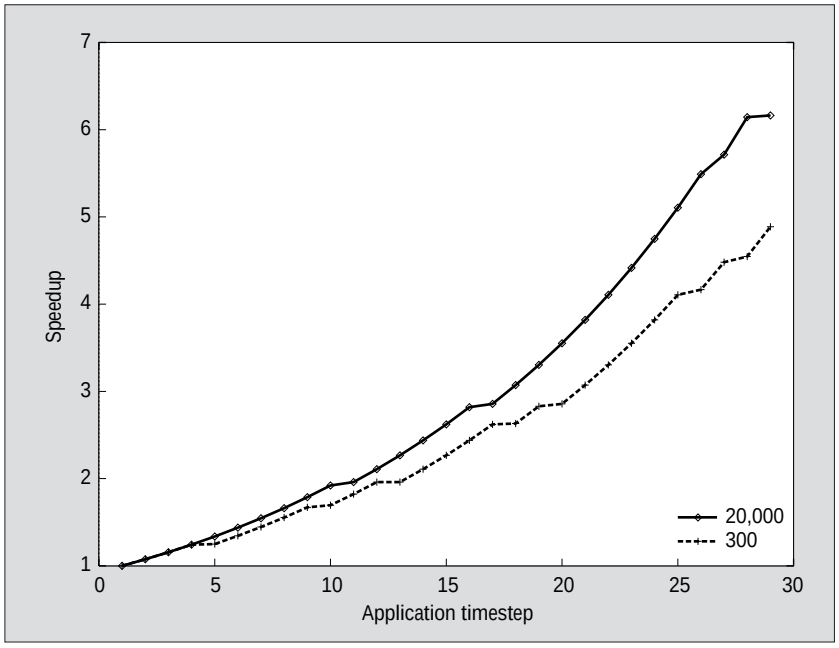

Figure 9. Speedup versus application time step. The heat-diffusion simulation executes on 10 threads with an unbalanced initial distribution. As the simulation executes, the steering system attempts to balance the load. A higher steering frequency of 20,000 Hz provides a quicker initial balance than does the lower setting of 300 Hz.

To test the effects of a slower system, we induced a latency in the decision process to lower the steering frequency to approximately 300 Hz. Although both applications eventually achieved a reasonable load, the higher steering frequency of 20,000 Hz achieved balance at about 10 time steps before the lower frequency of 300 Hz did. This frequency is unimportant in later time steps when the load balance becomes stable.

Interestingly, the lower frequency of 300 causes a mismatch between the sensor events and the arming of actuators, as Figure 10 shows. If the time between event generation and actuator arming is too long, the actuator cannot inject its requested change until some later time step—essentially losing the steering opportunity for that time step. Nevertheless, it is important that the steering cycle $\Delta_{steercycle}$ be fast relative to the application to prevent these missed steering opportunities.

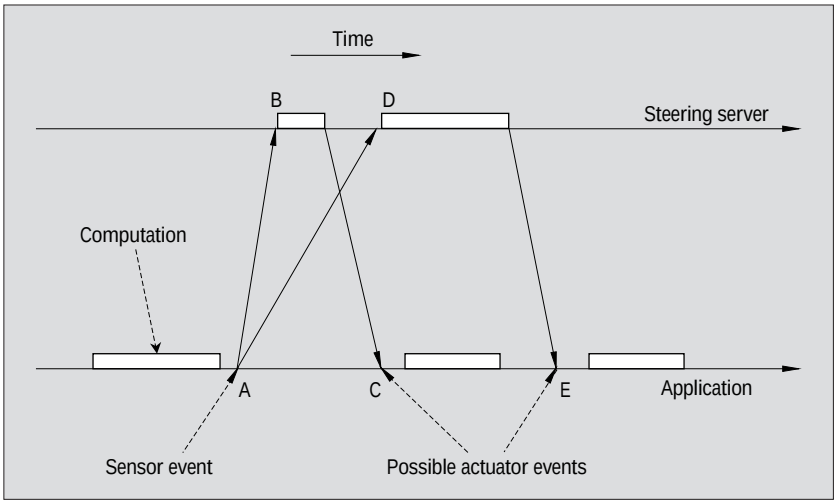

Figure 10. Missed steering opportunities. If the lag time from the generation of the monitoring data to the enactment of the steering command is too long, the actuator might have to wait until later opportunities to make its change.

OUR INITIAL experience with Magellan has been positive, and we have met most of the major design criteria. However, several improvements would help this approach to computational steering. First, ACSL cannot optimize all commands. In some cases, the user's steering request demands brute-force tactics to retrieve data and steer the system. Second, although it is clear that application-specific visualizations and steering can be very useful, generic tools must be developed that help users identify steering opportunities in the code. For example, some applications have thousands of variables and procedures. Even with advanced visualization techniques, forcing the user to compare and correlate these items is unattractive. Finally, it is also clear that the system must allow more complex forms of steering. Our language provides a basis for these abstractions, but more work is necessary.

Although ACSL is specified with an LALR(1) grammar, it is congruent with graphical user interfaces for steering. GUIs can use the language to control the steering system. Although ACSL provides a technique for steering applications, it does not enforce any policies and, by design, forces developers to create their own policies. Analysis of policies for computational steering remains an open area. Each application can have thousands of parameters, and various application experts must apply application steering to their own domains to determine the most appropriate policies for computational steering. We are currently experimenting with large-scale applications and simple steering policies. We expect that such experimentation will result in a richer set of ACSL constructs and additional optimizations for sets of ACSL statements.

Acknowledgments

While Jeffrey Vetter was a PhD candidate at the Georgia Institute of Technology, NASA financially supported him with a Graduate Student Researchers Program Fellowship. Additionally, this work was funded in part by Los Alamos National Laboratory and in part by NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846.

References

1. J. Leech, J.F. Prins, and J. Hermans, "SMD: Visual Steering of Molecular Dynamics for Protein Design," *IEEE Computational Science & Eng.*, Vol. 3, No. 4, Winter 1996, pp. 38–45.
2. D.J. Jablonowski et al., "VASE: The Visualization and Application Steering Environment," *Proc. Supercomputing '93*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1993, pp. 560–569.
3. E. Kraemer and J. Wallis, "Visualization and Interactive Steering of Simulated Annealing," *Proc. SPDP '96 Workshop Program Visualization and Instrumentation*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996.
4. J.S. Vetter, "Computational Steering Annotated Bibliography," *SIGPLAN Notices*, Vol. 32, No. 6, June 1997, pp. 40–44.
5. J. Vetter and S. Schwan, "Progress: a Toolkit for Interactive Program Steering," *Proc. 24th Int'l Conf. Parallel Processing*, Vol. 2, CRC Press, Boca Raton, Fla., 1995, pp. 139–142.
6. D.M. Ogle, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 7, July 1993, pp. 762–778.
7. V.E. Taylor et al., "Interactive Virtual Reality in Simulations: Exploring Lag Time, *IEEE Computational Science & Eng.*, Vol. 3, No. 4, Winter 1996, pp. 46–54.

Jeffrey Vetter is a computer scientist at the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory. His research interests are mainly experimental software systems for high-performance computing: advanced software-development environments, automated data exploration, visualization, and ASCI-scale application development and analysis. Currently, his projects include the design and evaluation of scalable techniques for performance analysis on ASCI systems. He earned his BS in electrical engineering and his MS in computer science, both from Georgia Tech, Atlanta, and his PhD in computer science from the Georgia Institute of Technology. Contact him at LLNL, Center for Applied Scientific Computing, Lawrence Livermore Nat'l Laboratory, Box 808, L-661, Livermore, CA 94551; j-vetter@llnl.gov; www.llnl.gov/CASC/people/vetter.

Karsten Schwan is a professor in the Systems Research Group of the College of Computing at the Georgia Institute of Technology. He has been working with experimental and commercial high-performance parallel and distributed machines since their inception in the late 1970s—first addressing scientific applications and their operating and programming system needs, and then considering real-time applications and the operating system support they require. His research continues to be driven by large-scale applications developed jointly with end users, typically in the context of shared distributed laboratories. He received a BS in mathematics and computer science from Christian-Albrechts University, Kiel, West Germany, and both an MS and a PhD in computer science from Carnegie Mellon University. Contact him at the College of Computing, Georgia Inst. of Technology, Atlanta, GA 30332; schwan@cc.gatech.edu; www.cc.gatech.edu/fac/Karsten.Schwan.