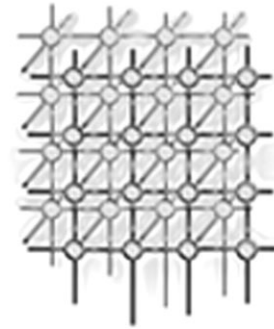# Performance technology for parallel and distributed component software

A. Malony[1,*,†], S. Shende[1], N. Trebon[1], J. Ray[2], R. Armstrong[2], C. Rasmussen[3] and M. Sottile[3]

[1]*Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, U.S.A.*
[2]*Sandia National Laboratory, Livermore, CA 94551, U.S.A.*
[3]*Los Alamos National Laboratory, Advanced Computing Laboratory, Los Alamos, NM 87545, U.S.A.*

SUMMARY

This work targets the emerging use of software component technology for high-performance scientific parallel and distributed computing. While component software engineering will benefit the construction of complex science applications, its use presents several challenges to performance measurement, analysis, and optimization. The performance of a component application depends on the interaction (possibly nonlinear) of the composed component set. Furthermore, a component is a 'binary unit of composition' and the only information users have is the interface the component provides to the outside world. A performance engineering methodology and development approach is presented to address evaluation and optimization issues in high-performance component environments. We describe a prototype implementation of a performance measurement infrastructure for the Common Component Architecture (CCA) system. A case study demonstrating the use of this technology for integrated measurement, monitoring, and optimization in CCA component-based applications is given. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: component software; performance; parallel; distributed; optimization

1. INTRODUCTION

The power of abstraction has played a key role throughout the history of scientific computing in managing the growing complexity of scientific problem solving. While the evolution of software abstractions and the technology that supports them has helped to address the challenges of scientific

---

*Correspondence to: A. Malony, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, U.S.A.
†E-mail: malony@cs.uoregon.edu

application development, it has at times been in conflict with the ability to achieve high performance. On the one hand, abstraction in software systems further distance the scientific application developer from the range of sources of performance behavior and possible performance problems. On the other hand, support for performance observation and analysis has been poorly integrated in software systems, making performance evaluation a more difficult task. As both the power and the complexity of scientific software environments and computing systems mutually advance, it is imperative that technology for performance evaluation and engineering keep pace and become part of an integrated software and systems solution.

The software challenges of building large-scale, complex scientific applications are beginning to be addressed by the use of component software technologies (see, e.g., [1–3]). The software engineering of scientific applications from components that can be 'plugged' together will greatly facilitate construction of coupled simulations and improve their cross-platform portability. However, the success of scientific component software will depend in the end on the ability to deliver high-performance solutions. Scientific components are more complex and diverse than typical software components or libraries, in their scale, execution modes, programming styles and languages, and system targets. Performance technology that lacks robustness, portability, and flexibility will inevitably prove incapable of addressing the software and platform integration requirements required for performance observation and analysis. *Intra-component performance engineering* addresses problems of individual component performance characterization, analysis, modeling, and adaptation. *Inter-component performance engineering* provides local and overall awareness of application performance and facilities to access that performance information for the application to utilize. Most importantly, performance engineering technology should be compatible with the component engineering methodologies and frameworks used to develop applications, or it will be neither routinely nor effectively applied by component and application developers.

Our research work on performance technology for component software is defined by three objectives. The first is a methodological and operation model for intra- and inter-component performance engineering. Here, we define how the performance engineering technology will integrate with the component architectures and frameworks being considered for scientific computing. The Common Component Architecture (CCA) Forum [4] is specifying component software extensions and infrastructure to address problems of parallel component interfaces, scientific data exchange, and cross-language interoperability. We chose the CCA specification as our reference archetype. The second objective is the development of technology to implement the methods and techniques required for intra- and inter-component performance engineering in the context of existing scientific component efforts. Our target audience for the performance engineering technology we are creating are the framework and application developers using the CCA specification. Specifically, we are integrating our TAU performance system [5] with the CCA software, such as the Scientific Interface Definition Language (SIDL) [6], the Babel component interface toolkit [7], and the CCAFFEINE framework [1]. Our final objective is the application of the model and technology for performance engineering to real component-based scientific computing environments. The goal here is to demonstrate both the capability and utility of our performance engineering ideas.

The three objectives above are covered in this paper. Section 2 discusses the use of component technology for scientific computing and motivates the general requirements for performance engineering. The functional operation of the CCA specification is also described. Our conceptual model for performance engineering of component software is presented in Section 3. Here we describe

a high-level methodology to technology development for intra- and inter-component performance engineering. Section 4 then considers the implementation of this technology in the context of the CCA software environment. Section 5 presents an application of our work with an emphasis on CCA performance modeling and optimization. Here we demonstrate the use of a CCA performance interface in measurement experiments to construct empirical performance models. We also show how, when the application runs, an optimizing component utilizes the performance application programmer interface (API) to gather statistics about the running application and decides which of the sets of similar components to choose for optimal performance.

Our work has been targeted primarily towards high-performance computing (HPC) environments. However, it naturally extends to Grid computing both in terms of application of component technologies and the requirements for performance engineering. We briefly discuss the extensions and the issues that arise as part of the conclusions Section 6.

2.   COMPONENT TECHNOLOGY FOR SCIENTIFIC COMPUTING

Component technology extends the benefits of scripting systems and object-oriented design to support reuse and interoperability of component software, transparent of language and location [8]. A *component* is a software object that implements certain functionality and has a well-defined interface that conforms to a component architecture defining rules for how components link and work together. The term *component framework* is used to designate a specific component architecture implementation. Component technology offers many advantages to scientific computing since it allows domain-level knowledge to be encapsulated in component building blocks that can easily, hopefully efficiently, be used in application development, removed from the concerns of how the component was developed or where it resides. As a result, scientists can focus their attention to overall application design and integration.

Unfortunately, the three most widely-used component standards (CORBA [9], COM/DCOM [10], Java Beans [11]) are ill-suited to handle high-performance scientific computing due to a lack of support for efficient parallel communication, insufficient scientific data abstractions (e.g. complex numbers), and/or limited language interoperability [12]. Furthermore, often the software does not run on the systems scientists use, or simply runs too slow for their applications.

2.1.   CCA

To overcome some of the limitations of standard component software approaches for scientific computing, the CCA Forum [4] was started in 1997 to define the standard foundations of scientific component architecture and to facilitate the adoption of CCA tools and technologies. In addition, the U.S. Department of Energy (DOE) established the Center for Component Technology for Terascale Simulation Software (CCTTSS) [13] for purposes of developing the CCA software infrastructure and demonstrating its use for complex scientific simulations.

Component programming, much like object-oriented programming, provides a model for constructing software such that units of code (components and objects) expose a 'public' interface to the outside while hiding their internal implementation features. Components extend the object model by allowing components to dynamically discover and expose interface information, something that is
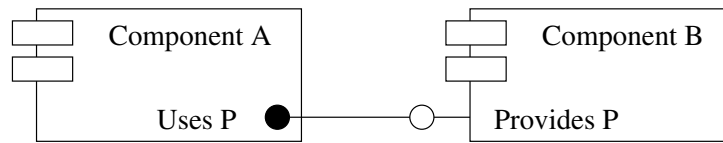
Figure 1. Two CCA components: One *uses* a 'P' port *provided* by the other.

statically determined at compilation time in most object-oriented languages. Fundamentally, the CCA is a specification of the component programming pattern and the interface the components see to the underlying support substrate, or *framework*. The CCA allows components to describe their interfaces in the Scientific Interface Definition Languaged (SIDL) [6]. Like the IDL used by CORBA, the interfaces are defined in a language independent manner and are not bound to the source code or compiled binary of a component. The IDL simply describes the public interface so that external parties can discover what services are available and how they must be called.

In the CCA, a component is defined as a collection of *ports*, where each port represents a set of functions that are publicly available. A port is described using SIDL, and some form of wrapper exists in the implementation to map the SIDL interface to that of the implementation language. From the point of view of a component, there are two types of ports. Those that are implemented by a component are known as *provides* ports, and other components may connect to and use them. Other ports that a component will expect to be connected to and call are known as *uses* ports. Uses and provides ports are connected together as shown in Figure 1. The act of connecting components is referred to as component *composition*.

When a component is instantiated and allowed to execute, it registers the provides and uses ports with the underlying framework. This information allows external components to discover what ports or interfaces are available, and ensures that expected relationships between components are fulfilled, before allowing execution. Port discovery is a service provided by the framework and is actually just another port that a component can connect to. For instance, a component can obtain a list from the framework of all components providing a specific interface or port. The component could then connect to each of the ports in the list in an iterative fashion and make calls on the methods within the connected port. Iterative access to a common set of interfaces presupposes that such a set exists. One of the primary benefits of component-based programming is the adoption of common interfaces for domain-specific purposes. This allows different teams of individuals to develop components based on this 'standard' component API. This, in turn, allows users of these components to pick and choose the particular component that best fits their needs.

The CCA architecture is the operational foundation for the implementation of CCA-compliant component frameworks, but it does not by itself directly specify how requirements of high-performance component systems are met. Rather, the intent of the CCA specification is to allow the preservation of performance. This takes four primary forms. First, the performance of the functional 'core' of a component (e.g. linear equation solver) should not suffer because it is implemented within a component. In general, this demands support for all core languages (e.g. C, C++, and Fortran 77, 90 and 95), and support for all forms of parallel execution that the core software may employ.

Second, component communication mechanisms should not be dictated but be selectable based on what best suits encapsulated functionality and component–component proximity (e.g. same memory space, same cluster, local network or wide-area). Third, parallelism between components should not be restricted. This regards primarily support for parallel communication between components, but also relates to parallel component computing paradigms, as in the *Single Component Multiple Data (SCMD)* model [1], *MxN coupling* [14], and the macro-level dataflow *Uintah Computational Framework (UCF)* [15]. Lastly, it should be possible to select component instances and configure component compositions for performance purposes, both prior to and during execution.

2.2. Performance engineering and component software

The above important performance-related aspects of component-based scientific computing motivate fundamental performance engineering questions that are examined specifically in our research.

- How are the performance of components and component compositions evaluated?
- How can this evaluation be done robustly given the diversity of component (component composition) types and implementations?
- How are component performance data represented and made available to the component framework as a whole?
- How can the performance of component compositions be modeled?
- What restrictions does a component approach place on performance engineering, and can the component architecture, framework infrastructure, and technologies be leveraged to implement and deliver performance engineering support more effectively?
- Will the integration of performance engineering technology lead to a quantifiable improvement in performance of component-based scientific computing and how will this be demonstrated?
- Is it possible to develop a component performance engineering solution that is general purpose and can be applied in many scientific computing contexts?

We address these questions from the perspective of the technology to deliver performance-engineered solutions for scientific computing environments.

2.3. Related work

Since commercial component models are targeted mostly at serial computing environments, the design of performance methods and metrics for these software systems are not likely to account for critical requirements important to high-performance scientific computing such as memory hierarchy performance, data locality, or floating point operation. Likewise, the distributed frameworks/component models (e.g. DCOM and CORBA) use commodity networking to connect components together, and lack consideration for high-performance network communication required for HPC. In a distributed environment, metrics such as round-trip time and network latency are often considered useful, while quantities such as bisection bandwidth, message-passing latencies and synchronization cost, which form the basis of much of the research in scientific performance evaluation, are left unaddressed.

However, despite the different semantics, several research efforts in these standards offer viable strategies in measuring performance. A performance monitoring system for the Enterprise Java Beans standard is described in [16]. For each component to be monitored, a proxy is created using the same interface as the component. The proxy intercepts all method invocations and notifies a monitor component before forwarding the invocation to the component. The monitor handles the notifications and selects the data to present, either to a user or to another component (e.g. a visualizer component). The goal of this monitoring system is to identify hot spots or components that do not scale well.

The Wabash tool [17,18] is designed for pre-deployment testing and monitoring of distributed CORBA systems. Because of the distributed nature, Wabash groups components into regions based on the geographical location. An interceptor is created in the same address space of each server object (i.e. a component that provides services) and manages all incoming and outgoing requests to the server. A manager component is responsible for querying the interceptor for data retrieval and event management.

In the work done by the Parallel Software Group at the Imperial College of Science in London [19,20], the research is focused on Grid-based component computing. However, the performance is also measured through the use of proxies. Their performance system is designed to automatically select the optimal implementation of the application based on performance models and available resources. With n components, each having C_i implementations, there is a total of $\prod_{i=1}^{n} C_i$ implementations to choose from. The performance characteristics and a performance model for each component is constructed by the component developer and stored in the component repository. Their approach is to use the proxies to simulate an application in order to determine the call-path. This simulation skips the implementation of the components by using the proxies. Once the call-path is determined, a recursive composite performance model is created by examining the behavior of each method call in the call-path. In order to ensure that the composite model is implementation-independent, a variable is used in the model whenever there is a reference to an implementation. To evaluate the model, a specific implementation's performance model replaces the variables and the composite model returns an estimated execution time or estimated cost (based on some hardware resources model). The implementation with the lowest execution time or lowest cost is then selected and a execution plan is created for the application.

3. PERFORMANCE ENGINEERING METHODOLOGY

Building efficient scientific applications as hierarchical compositions of cooperating components depends significantly on having a thorough knowledge of component performance and component interactions. To make effective decisions concerning component configuration, deployment, and coupling, it is important to develop a *performance engineering methodology* that complements the component and application development and execution processes. Ideally, the methodology would be supported by performance technology (measurement, analysis, and modeling) that extends the programming and execution environment to be *performance observable* and *performance aware*. However, the diversity of component functionality and the complexity of component implementation (e.g. different languages, hardware platforms, and parallelism modes) challenge performance technology to offer (more) robust solutions. Our objective here is to define a consistent performance engineering model for components and component ensembles, and to implement that model in an integrated fashion throughout the component development and execution framework.
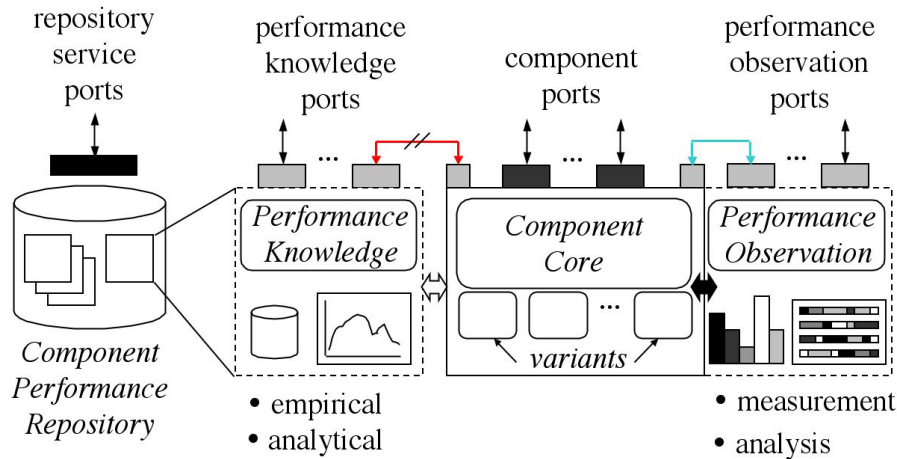
Figure 2. Performance engineered component design.

3.1. Component performance engineering

While component architectures facilitate the development of complex applications by allowing the independent creation of generic, reusable components and by easing their composition, application-level performance engineering requires knowledge about a component's performance in different contexts and observation of the component's performance behavior during execution. We view a *performance-engineered component* as having four constituent parts:

- performance characterization and modeling (*performance knowledge*);
- integrated performance measurement and analysis (*performance observation*);
- runtime access to performance information (*performance query*);
- mechanisms to alter component performance behavior (*performance control*).

In Figure 2, we show how a component's design may be extended to support these performance engineering features. The intent is to keep the extended (performance engineered) design relatively consistent with the CCA model.

We represent a component's generic architecture by the middle (solid) box in the figure. As shown, we distinguish between a component's 'core' functionality and its variant sub-parts (both functional variants and code variants) that may be selected prior to or during component execution‡.

---

‡This view is consistent with how components are regarded in several systems. (Note, a component's ports can also have variants, but this is not drawn to reduce diagram complexity.)

A component's generic design can be 'performance engineered' with support for *performance knowledge* and/or *performance observation*. Performance knowledge extensions (left (dashed) box) provide means to describe and store what is 'known' about the component's performance. This can take the form of empirical characterizations in a performance database, as well as performance models captured from empirical studies or analytical performance analysis. Performance observation extensions (right (dashed) box) implement performance measurement and analysis capabilities that are used during component execution. In addition, support for querying the performance information and for effecting control based on performance feedback complement the performance knowledge and observation parts.

To justify these extensions, let us first consider the use of performance knowledge in the performance engineering of component frameworks. The ability to save information about a component's performance allows that knowledge to be used for performance-guided component selection, deployment, and runtime adaptation. However, the representation of performance knowledge must be in common forms and there must be standard means to access selective performance information. In Figure 2, we identify a 'component performance repository' where the performance knowledge could be kept and queried within the component framework, similar in concept (and, likely, function) to the CCA component repository. In fact, we could view the performance knowledge extension as a component in its own right, as suggested in the figure by the separation of the left (dashed) box from original component. In this way, the *performance knowledge component (PKC)* could provide PKC ports that give component-level access to the performance information both to other components within the framework (black arrowhead) as well as back to the original component (shaded arrowhead) whose performance it represents. These 'feedback' port connections would allow for both static (instantiation-time) and dynamic (runtime) component control. The PKCs could also be instantiated and used, in particular, as active components to build runtime composite performance models for resource allocation and scheduling.

Similarly, let us consider justification for performance observation support in our performance engineered component model. The ability to observe a component's execution time performance is important for two reasons. By definition, any empirically derived performance knowledge requires some form of performance measurement and analysis. However, it does not mean that this support necessarily must be integrated in the component's design. The second reason is to monitor the performance of a component during execution and use that information in dynamic performance decision making. Here, integrated performance observation support (measurement and analysis) is key. As depicted in Figure 2 (block-filled double arrow), this integration requires component instrumentation (both core and variant), runtime measurement and data collection, and online and/or offline performance analysis. In this respect, we can view performance observation in the performance engineered component model as a functional extension of the original component design. To allow the component framework to query performance behavior, this extension could include new component methods and ports (black arrowhead). It is also useful for the original component to be able to access its own performance observations to make internal performance decisions. An interesting design issue here concerns how this functionality should be supported. Our view is to further generalize the model to regard observation support as encapsulated in a *performance observation component (POC)* that is tightly coupled and co-resident with the original component. Special POC 'provides' ports allow the original component to use optimized interfaces (shaded arrowhead) to access 'internal' performance observations.
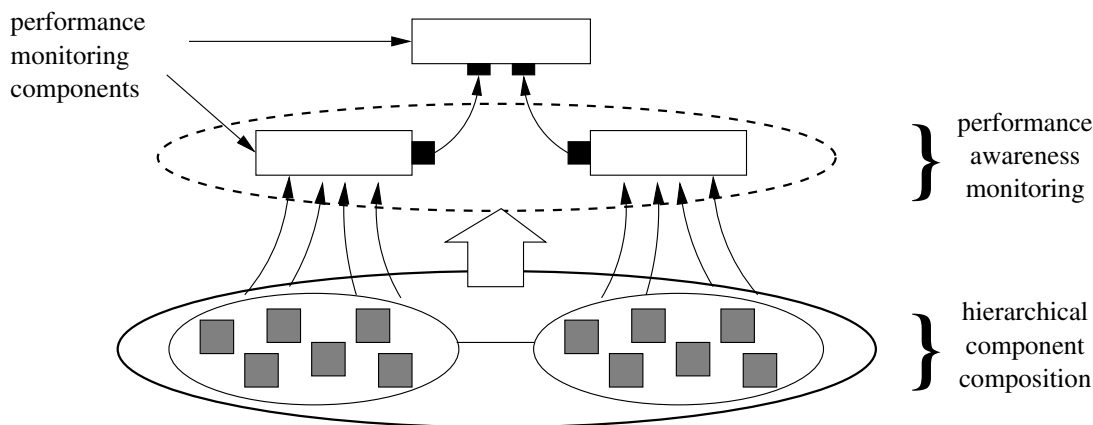
Figure 3. Hierarchical performance awareness monitoring for component composition.

3.2. Component composition performance engineering

Our discussion above focused on the performance engineering model at the component level. Now we turn attention to component composition. The performance of component-based scientific applications depends as much (perhaps more) on the interplay of component functions and the computational resources used for component execution and interoperation, as it does on individual component performance. Because component operations and interactions can be complex and dynamic, and resource availability may be subject to variation, the management of component compositions throughout their execution becomes a critical element of their successful deployment and use. In general, this management can include resource assignment, component placement and scheduling, data communications control, and file storage and allocation. However, the issue for us is not necessarily to design the specific tools that will be part of the performance management solution, but to identify key technological capabilities needed to support the performance engineering of component ensembles. In practice, this distinction is blurred, admittedly due to the different infrastructure used for scientific component computing, ranging from wide-area Grids of heterogeneous resources to tightly-coupled large-scale computing systems. Thus, we restrict our attention to only two model concepts: *performance awareness* and *performance attention*.

Performance awareness of component compositions relates to an ensemble view of performance, how this information is obtained, and how it can be accessed. As with components, performance engineering looks at composition performance knowledge and observation. Composition performance knowledge can come from empirical as well as analytical evaluation, can utilize information provided at the component level, and can be stored in repositories for future review. Performance awareness extends the notion of component observation to *ensemble-level performance monitoring*. The idea is to associate monitoring components with levels of hierarchical component grouping, building upon component-level observation support. This is presented in Figure 3. These monitoring components

act as performance integrators and routers, using component framework mechanisms to connect performance data 'producers' to performance data 'consumers'. Knowledge and observation support at the ensemble level can work together to evaluate performance goodness of an application during execution by comparing performance achieved to performance expected. Also, a global performance model, constructed from component models, could be used to make intelligent decisions for runtime crafting of an application to a particular computing environment, guided by performance observations.

Performance attention is the notion that there are active elements (*sensors*) in a component application looking for particular performance *behaviors* or *patterns*. These elements utilize access to component performance data to decide when performance phenomena are significant. At these instances, performance attention mechanisms may signal performance events that then notify other parts of the application about what was detected. Decisions about what to do in response to these performance events can then be made. Performance attention is utilized within an application to help control, adapt, or steer its performance. Component framework support, such as event services, can be leveraged to develop performance attention technology.

4.   DEVELOPMENT APPROACH

Based on the methodology presented above, this section presents an approach to the development of performance engineering technology for scientific component software. Our focus is on the component software and frameworks being developed in the DOE CCTTSS [13]. Thus, our work is based on the CCA specification [1,12] and integrated with the existing CCA technologies. We describe our development approach in respect to five areas[§]:

- component performance instrumentation and measurement;
- performance of component connections;
- performance monitoring;
- component performance database and knowledge repository;
- component performance modeling.

4.1.   Component performance instrumentation and measurement

By design, component software provides standard interfaces for use within a component system, but hides details of component implementation. Thus, it should be expected that components will be built using different programming languages and may run on different system platforms. This is true even for components that implement the same functionality. Furthermore, scientific components will be implemented to run in high-performing modes, including parallel execution. In fact, for CCTTSS applications, the components used may be scalable to large size (problem and number of processors). One basic challenge to component performance engineering is how to measure component performance.

[§]The first three areas were initially discussed in [21].

4.1.1. TAU *performance system*

The diversity of component implementation demands a robust parallel performance measurement system. The TAU project has developed performance technology for complex parallel and distributed systems based on a general complex systems computation model and a modular performance observation and analysis framework. It targets a general computation model consisting of shared-memory computing *nodes* where *contexts* reside, each providing a virtual address space shared by multiple *threads* of execution. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Because TAU enables performance information to be captured at the node/context/thread levels, this information can be mapped to the particular parallel software and system execution platform under consideration.

The TAU performance system [5] supports a flexible instrumentation model that applies at different stages of program compilation and execution. The instrumentation targets multiple code points, provides for mapping of low-level execution events to higher-level performance abstractions, and works with multi-threaded and message-passing parallel computation models. Instrumentation code makes calls to TAU's measurement API. The measurement library implements performance profiling and tracing support for performance events occurring at function, method, basic block, and statement levels during execution. Performance experiments can be composed from different measurement modules (e.g. hardware performance monitors) and measurements can be collected with respect to user-defined performance groups. The TAU data analysis and presentation utilities offer text-based and graphical tools to visualize the performance data [22] as well as bridges to third-party software, such as Vampir [23] for sophisticated trace analysis and visualization.

4.1.2. *A performance interface for components*

Given the TAU performance measurement technology, the important question becomes what is the approach best suited for component performance measurement. There are two measurement types we envision based on how a component is instrumented: (1) with direct calls to a measurement library; or (2) using an abstract measurement interface. The difference is depicted in Figure 4. TAU specializes in multi-level performance instrumentation targeting a common performance measurement API. We can instrument the component code to call TAU measurement routines directly using the API, as shown in the left part of Figure 4. To facilitate the instrumentation, TAU provides automated source instrumentation tools (based on the Program Database Toolkit (PDT) [24]) and dynamic instrumentation support. As shown, the TAU measurement system maintains runtime performance data that can be accessed via the API directly or stored in files at the end of component execution.

In contrast, the component could also be instrumented to call an abstract measurement component interface, as shown in the right part of Figure 4. This interface would be implemented by a *performance component* that targets a backend measurement system (in this case TAU). There are several benefits to this approach. First, a component could be developed with 'virtual instrumentation' in the sense that the abstract measurement interface is virtual (i.e. consists of virtual functions). The overhead of instrumentation is nullified until a performance component is instantiated. Second, it is possible to use any measurement system in the performance component that can conform to the interface. Lastly, the performance component can provide ports for other components to use, including ports to access performance data without touching the instrumented application component. This raises the possibility
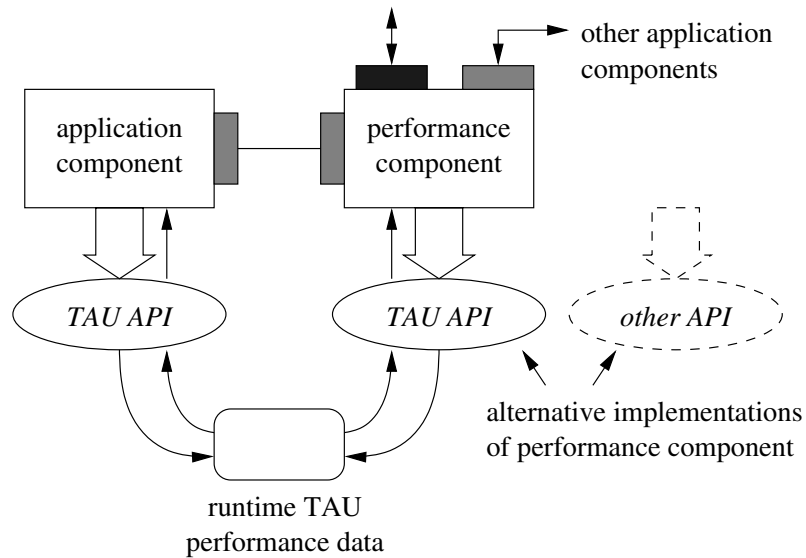
Figure 4. Measurement component interface using TAU .

that the application component is instrumented directly, but the performance data are accessed via the performance component. The downside of this approach is that the measurement interface is possibly less efficient or that it does not allow certain types of detailed performance measurements to be made.

4.1.3.   A TAU *performance component*

Our approach offers both types of measurements discussed above. In particular, we have designed a performance instrumentation interface for component software and a performance component that implements this interface through a measurement port. This interface allows a user to create objects for timing, track application events, control the instrumentation at runtime, and query the performance data. TAU provides an implementation for each of these entities.

Timer interface

A timer interface allows the user to bracket parts of their code to specify a region of interest. The `Timer class` interface supports the `start` and `stop` methods. A timer object has a unique name and a signature associated with it. There are several ways to identify timers and performance tools have used different techniques. To identify a timer, one approach advocates the use of numeric identifiers and an associated table mapping the identifiers to names. While it is easy to specify and pass the timer identifier among routines, it has its drawbacks. Maintaining a table statically might work for languages such as Fortran 90 and C, but it extends poorly to C++, where a template may be instantiated with different parameters. This aspect of compile time polymorphism makes it difficult to

disambiguate between different instantiations of the same code. Also, it can introduce instrumentation errors in maintaining the table that maps the identifiers to names. This is true for large projects that involve several application modules and developers.

Our interface uses a dynamic naming scheme where timer names are associated with the timer object at runtime. A timer can have a unique name and a signature that can be obtained using runtime type information of objects in C++. Several logically related timers can be grouped together using an optional profile group. A profile group is specified using a name when a timer is created. TAU implements a generic Timer interface and introduces an optimization that allows it to keep track of only those timers that are invoked at least once. It maintains both exclusive and inclusive measurement values for each timer. Timers can be nested, but may not overlap (i.e. start and stop calls from one timer should not overlap those from another). When timers overlap, TAU detects this overlap at runtime and warns the user about this error in instrumentation.

It is important to note that this interface is independent of the nature of measurements that can be performed by the performance tool. For instance, TAU may be configured to record exclusive and inclusive wallclock time for each timer for each thread of execution. Other measurement options that are currently supported include profiling with process virtual time or counts obtained from hardware performance counters. TAU also provides the option of making multiple measurements using a combination of wallclock time and/or hardware performance metrics in the same performance evaluation experiment. Thus, the timer interface is independent of the underlying measurements and is a vehicle for the user to specify interesting code regions that merit observation.

Control interface

The control interface provided by the performance component allows us to enable and disable a group of timers at a coarse level. The user can disable all the groups and selectively enable a set of groups for refining the focus of instrumentation. Or the user can start with all groups in an enabled state and selectively disable a set of groups.

Query interface

The query interface allows the program to interact with the measurement substrate by querying for a variety of performance metrics. This interface allows the program to query the set of measurements that are being performed. These are represented in the interface as a list of counters. The query interface reports the list of timers that are active at any given point in time. For each timer, it provides a set of exclusive and inclusive values for each counter. It provides the number of start/stop pairs (referred here as the number of calls) for each timer and also the number of timers that each timer called in turn. Instead of examining this data at runtime, an application may choose to store this information in files. This data may be read by an online monitor external to the application and analyzed as the application executes.

Event interface

The event interface provided by the performance component allows a user to track application-level events that take place at a specific location in the source code (as opposed to bracketing the code with start/stop calls). The generic event interface provides a single trigger method with a data parameter.

This permits the user to associate the application data with the event. For example, to track the memory utilization in an application, a user may create a named event called 'Memory used by arrays' and each time an array is allocated, this event might be triggered with the size of the chunk of memory allocated as its parameter. TAU implements the event class by keeping track of maxima, minima, mean, standard deviation, and number of samples as statistics. Another tool might, for instance, maintain quantiles for the same data.

The performance component interface gives each tool the flexibility of performing tool-specific optimizations, measurement and analysis unique to the tool, and provides a balance between tool specificity and genericity. For example, a tool may implement the `Timer` and `Event` classes in different ways. The benefits of such an interface are manifold for a user. Using this generic interface to annotate the source code, the user can benefit from using multiple performance measurement and analysis tools without the need for recompiling the source code. At runtime, the user can choose which tool (and more specifically, which dynamic shared object) implements the interface and instantiates a component for performing the instrumentation. This approach permits the user to mix and match the capabilities of multiple performance tools to accomplish the task of performance observation of components.

4.2. Performance of component connections

The type of component connections and the behavior of component interactions are equally as important to the overall performance of an application as the performance of individual components. This is true both to understand the frequency and granularity of component interactions, but also the degrees and type of data exchange that occurs. Thus, we must have support to observe component connections and how they are used. Our approach to this problem is to integrate performance instrumentation in the process of interface definition and generation. This instrumentation will target measurement mechanisms built with TAU to capture performance information specific to interface type, operation, and invocation.

The main problem we need to address is how component interfaces are instrumented. We assume that the component interface is CCA-compliant and, therefore, follows the methodology of defining provides and uses ports. If the component interface is implemented directly in C++ (i.e. using the CCAFFEINE framework), we can use the mechanisms for component instrumentation discussed above, both manual and automatic. It is also reasonable to consider whether the CCA `classic::gov::cca::Port` class should be extended to facilitate instrumentation in this case.

However, if the component interface is specified in SIDL and generated with a SIDL compiler (i.e. Babel [7]), we have two choices to consider. One option is to wait until the interface code is generated and instrument it in a similar manner to how we use TAU to instrument the C++ component interface. The difficulty here is that Babel is able to generate interface code in the component's native language (e.g. C, C++, F77, F90/95). Thus, we must be able to apply TAU's cross-language instrumentation appropriately to the different native languages Babel supports. Knowing how the code that Babel generates is constructed will be important to producing efficient TAU instrumentation. We can also attempt to automatically generate the instrumentation at the time of SIDL compilation. This approach will possibly allow more efficient and optimized TAU instrumentation.

Our goal is to use the performance instrumentation to observe component interactions. These interactions occur through the services one component provides and others use. CCA defines

the port abstraction as a means to specify what services a component will provide and use. We speak of component composition as the act of connecting components; effectively the binding of provides and uses ports. The instrumentation idea we develop is to construct 'port wrappers' (or, more appropriately, 'service wrappers') to observe service 'entry' and 'exit' events. However, a question arises concerning whether the observation of component interactions is from the point of view of the service provider or the service user. Not only is this a question of where instrumentation gets placed, but it also relates to the meaning of the performance data measured at these points. If we consider component interface instrumentation in the service provider, we will be able to measure, for example, how often a service provided by this component is invoked, how long the service takes, and other information characterizing service type. Alternatively, if we put the instrumentation in the service user component, we will be able to determine, for instance, the execution profile across all services this component invokes, as measured in that component.

In programs with subroutine libraries, the contrast above reduces to the problem of observing 'callee'/'caller' performance. However, with CCA components and frameworks, the situation is more complex, both from the instrumentation point of view and with respect to performance observation. Ideally, we want to observe performance from both provider and user points of view, and we intend to develop support for both. In addition, we want to associate performance to specific component (port) bindings. The CCA framework interconnects components by binding compatible provides and uses ports, but the binding information is not available to the instrumentation in the component interfaces at the time the instrumentation is created. Thus, we must develop additional mechanisms to pass this knowledge to the interface instrumentation.

4.2.1. Performance wrappers and proxies

Our approach will be to build on TAU's dynamic performance mapping features. At the time port bindings are made by the CCA framework, it is possible to inform the port performance wrappers of the connection (i.e. user) identity. This can occur either at the user or provider interface, or both. With this information, a performance mapping can be created to associate measured performance data with the specific service invocation. In this way, it is possible to track performance relative to component bindings. Components using services can gain performance views specific to service instantiation, allowing them, for instance, to evaluate components for performance acceptability. Components providing services can identify heavy component users, perhaps allowing their resources to be better allocated and utilized. Overall observation of component bindings will also be important for component ensemble optimizations.

Another approach is to create a *proxy component* for each component the user wants to analyze, similar to what is done for monitoring Java Beans [16]. The proxy component shares the same interface as the actual component. When the application is composed and executed, the proxy is placed directly 'in front' of the actual component. Since the proxy implements the same interface as the component, the proxy intercepts all of the method invocations for the component. In other words, the proxy uses and provides the same types of ports that the actual component provides. In this manner, the proxy is able to snoop the method invocation on the provides port, and then forward the method invocation to the component on the uses port. In addition, the proxy can use the performance interface to make measurements. If the method is one that the user wants to measure, monitoring is started before the method invocation is forwarded and stopped afterwards. The parameters that influence the method's

performance can be associated with the observed performance. This performance differentiation would allow for optimizations to be made by someone with knowledge of the algorithm implemented in the component. Creating a proxy from a component's header file is relatively straightforward. Currently, proxies are created manually with the help of a few scripts, but it is not difficult to envision proxy creation being fully automated.

4.3. Performance monitoring

Online performance monitoring builds on runtime performance measurement mechanisms to provide support for accessing performance data during execution. This is important to provide feedback for runtime optimizations and performance steering. Our focus is on building the needed infrastructure to allow the performance data 'producers' in component applications to connect to and communicate with performance data 'consumers' wherever they may be in the component system. The key work here is to implement this infrastructure to be CCA-compatible and, indeed, to leverage the CCA framework and services so that the monitoring system can be configured in a flexible and robust manner.

The CCA performance monitoring system builds on the performance component described in Section 4.1. Performance components are instantiated to make measurements on behalf of other CCA components. The performance component also supports a service to access the current performance information. (Note, this can be a challenge if the component is running in parallel.) Depending on the execution relationship of the components in an application (e.g. components are executed as separate processes), which is related to the CCA framework used to create the application, there may be one or more performance components. Thus, performance information for the whole CCA application will be distributed across the various performance components in operation.

The CCA performance components can be used by one component for measurements and by another for performance data access. This scenario is made possible by the CCA ports mechanism and CCA framework. While the performance component design and operation is enough to provide application-wide monitoring capabilities, we believe it is important to consider a hierarchical monitoring architecture that can adjust to the range of CCA application types to be expected. Such an architecture for monitoring component composition (see Figure 3) will allow efficiencies in how performance data are communicated, analyzed, and filtered in the system.

Our approach is to develop a performance monitor component that can interface with performance measurement components to access their performance data, merge performance data from different streams, conduct various analysis functions on the data (including filtering and reduction), and make results available to other monitor or application components. The functionality of these monitor components is controllable by application developers. With such monitor components, it will be possible to construct a monitoring system best matched to the needs of the application. Also, it will make it possible to build online performance analysis and visualization tools that can interface easily with the monitoring infrastructure. Ultimately, we want to link the monitoring system with a composition performance model, so as to allow global optimization of component applications.

4.4. Component performance knowledgebase and repository

Performance engineering technology for scientific component software should take into account not only how performance is observed, but also what techniques and tools are needed to improve the

performance of component applications. In general, performance tuning is the process of diagnosing performance problems by comparing measured performance with expected performance, and then removing the inefficiencies based on an analysis of those performance properties of components and component ensembles best matching execution conditions. We define performance tuning in this way because there are many factors that can affect the performance of component applications. These include the performance characteristics of individual components, the performance effects related to component coupling, and the resources available for component execution. Building and executing component applications will be guided by a set of choices that take these factors into account. These choices represent a complex performance space. Our goal is to develop performance technology that can assist application developers and users in making judicious decisions that will lead to high-performance component codes.

The approach we take provides a means to store performance knowledge about components and component compositions, and to make that knowledge available to applications at the times of component selection, deployment, and replacement during execution. A performance knowledgebase (PerfKB) can be used to record performance data for components measured from multiple performance experiments. In addition, the PerfKB is able to store results from empirical data analysis in the form of statistical tables and regression models. Analytical models used to express component performance with respect to algorithm characteristics can also be represented in the knowledgebase.

Once we have a means to capture and store performance knowledge, we need to support access to this information in ways that are consistent with component technology. As part of our plan, we integrate the PerfKB in the CCA framework services used for component registration and discovery. The idea is to extend component repository services with interfaces to the PerfKB where queries can be made about component performance. This will allow smart choices of which components and versions to use during application construction based on the known system parameters. Inquires can also be made to discover appropriate choice of parameters for component execution (e.g. number of threads or processors). Work within the CCTTSS project is underway to define a XML file specification to store component information, and we will work with this effort to incorporate what is needed for performance.

However, we believe this is only part of the solution. To control or improve the performance of an application during execution (i.e. performance steering), we must be able to make decisions based on how well a component is performing with respect to what is expected. We can think of performance aware components, which are able to monitor their own performance (using the performance measurement mechanisms discussed above) and adapt their operation in response to the performance feedback, but this depends on knowing what to change to effect positive performance gain. The performance knowledge to make the decision must be packaged in such a way so as to be easily accessed and processed. Requiring the component to return to the performance repository is too coarse-grained. Furthermore, the steering decisions are only with respect to local component performance. For intelligent performance control at the level of component ensembles, it is necessary to conceive of representations of performance knowledge that can be efficiently and actively processed.

In this context, a PKC can be implemented to capture specific performance information about a deployed component and to provide access to that information during execution using standard CCA interfaces. In essence, the PKC functions as a performance estimation (prediction) service for its respective component, where the virtual interfaces are specialized for the component type. Performance aware components can be created using this approach. More importantly, the approach

makes it possible to build higher-level performance models by combining the PKCs of the components employed in an application. These composition performance models provide performance awareness at the application level and can be used to guide performance steering decisions. Their CCA-based implementation using PKCs means performance estimations will be able to be made efficiently and with the benefit of existing CCA framework services.

4.5.   Performance experimentation and modeling

To achieve high-performing CCA applications, it is critically important to characterize and understand component performance and the performance of component compositions. For the most part, component applications are constructed in a functional manner by connecting components based on their offered services. However, there are few opportunities for performance optimization without performance information. The goal of performance modeling is to build up performance knowledge in a form that can be applied in performance decisions regarding component selection and configuration, component connections, scheduling of components, online component adaptation, and so on.

The performance of individual components can be studied in a straightforward manner by conducting a series of performance experiments on expected target platforms, varying algorithm- or system-specific parameters. In this way, a performance measurement space is enumerated, reflecting observed performance for chosen parameter and platform settings. The main complication of this approach, with respect to scientific components, is the complexity of the component implementations. For instance, parallel components will require performance experimentation for different numbers of processors. Component performance may also be sensitive to problem size and data layout. System features, such as processor and memory architecture, can lead to significant performance differences between machines. An experimentation harness will help to conduct performance measurements and could use a performance database to store the results. Analysis tools are also needed to synthesize empirical models from the performance experiments. The models will characterize performance behavior across experimental parameters and will serve as performance estimators of points not sampled in the performance space.

Difficulties arise when attempting to characterize the performance of component compositions. In general, it should not be expected that the principle of superposition holds when two components are connected. That is, if you combine the 'best' performing variants of each component, the component composition might not perform best or even well. This may be due, for instance, to data structure mismatch or conflicting memory usage. The experimentation approach should then involve testing of component combinations to characterize their performance interactions. However, as the number of connected components increases, the performance space grows and the interactions can become more complex. Again, performance experimentation studies are important, but techniques will need to be developed to help narrow the performance search.

5.   APPLICATION OF COMPONENT SOFTWARE PERFORMANCE ENGINEERING

Our initial implementations of the performance interface and performance component described in Section 4 are now being applied in the development and evaluation of CCA applications. This is happening in two general ways. The first way is to observe the execution time of a component

application and its composite components for purposes of an overall performance evaluation. The robust nature of our technology affords the developers an convenient solution to this problem, one that is consistent with the component programming approach and integrated into the CCA framework. The second way builds on the first to incorporate performance modeling and optimization in the process of application development and execution. This section describes an example of using our component performance technology for this purpose.

5.1. Selection of optimal component-based solvers

A situation that arises frequently in scientific computing is that of selecting a solver that both meets some convergence requirement and also performs optimally (i.e. reaches a solution in the shortest period of time). In general, a solver may be optimal for a particular class of problems, yet behave poorly on others. Even for a given class of problems, the convergence behavior of a solver can be highly dependent on the data themselves. Thus, the choice of an 'optimal' solver is not as easy as it might at first seem.

Using the component performance API described in the previous section, the CCA framework easily allows one to test a set of solvers on a representative subset of a broader spectrum of data. The best performing of the solvers can then be used to operate on the full dataset. While this is relatively easy to do using components with standard interfaces, it is a much more onerous task without. One must maintain separate bodies of code (one for each solver interface) and compile and link these codes against separate solver libraries. Then scripts must be generated to run the tests, select the best performer, and finally to make the final run. The component development methodology allows us to incorporate such testing and optimization as part of standard practice.

5.2. Interacting shock wave simulation

With the general strategy above in mind, consider the simulation of the interaction of a shock wave with an interface between two gases. The code employs structured adaptive mesh refinement [25,26] for solving a set of partial differential equations (PDEs) called the Euler equations. Briefly, the method consists of laying a relatively coarse Cartesian mesh over a rectangular domain. Based on some suitable metric, regions requiring further refinement are identified, the grid points flagged and collated into *rectangular* children patches on which a denser Cartesian mesh is imposed. The refinement factor between parent and child mesh is usually kept constant for a given problem. The process is done recursively, so that one ultimately obtains a hierarchy of patches with different increase grid densities, with the finest patches overlaying a small part of the domain. The more accurate solution from the finest meshes is periodically interpolated onto the coarser ones. Typically, patches on the coarsest level are processed first, followed recursively by their children patches. Children patches are also processed a set number of times during each recursion.

Figure 5 shows the component version of the code developed using the CCAFFEINE [1] CCA framework. On the left is the ShockDriver, a component that orchestrates the simulation. On its right is AMRMesh that manages the patches. The RK2 component below it orchestrates the recursive processing of patches. To its right are States and EFMFlux, which are invoked on a patch-by-patch basis. The invocations to States and EFMFlux include a data array (a different one for each patch) and an output array of the same size. Both of these components can function in two modes—sequential
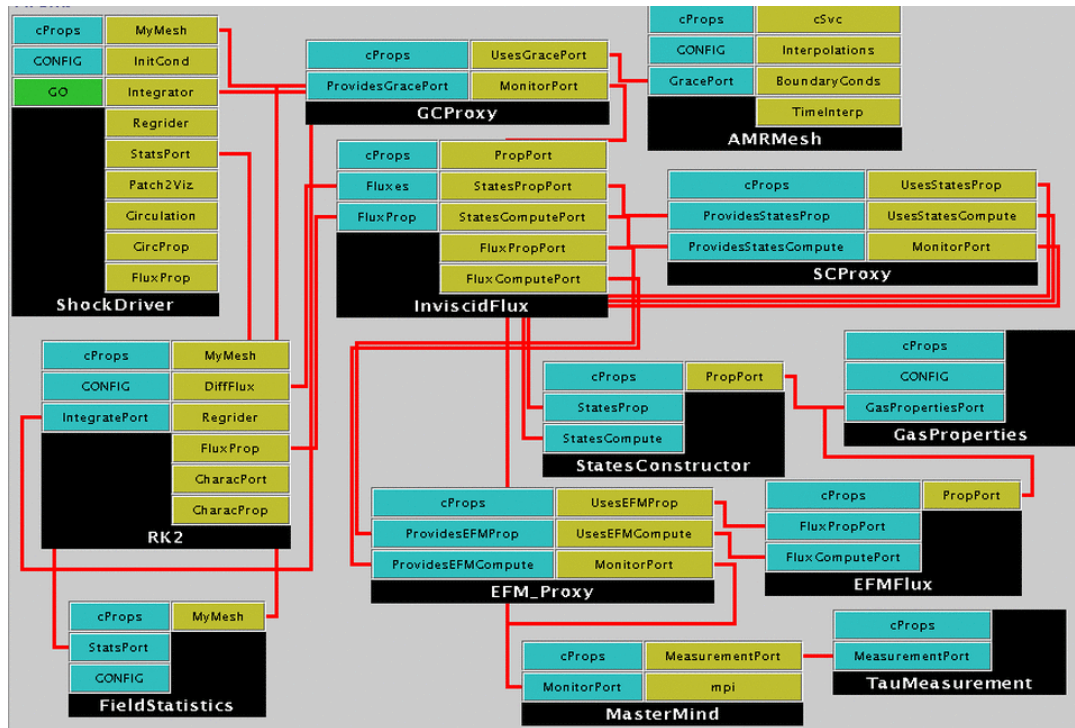
Figure 5. Snapshot of the component application, as assembled for execution. We see three proxies (for AMRMesh, EFMFlux and States), as well as the TAUMeasurement and Mastermind components to measure and record performance-related data.

or strided array access to calculate *X*- or *Y*-derivatives, respectively—with different performance consequences. Neither of these components involve message passing, most of which is done by AMRMesh.

We used the TAU performance interface and measurement component to model the performance of both States and EFMFlux and analyze the message-passing costs of AMRMesh. We also analyzed the performance of another component, GodunovFlux, which can be substituted for EFMFlux. Three proxies were created with performance interfaces and interposed between InviscidFlux and the component in question. A proxy was also written for AMRMesh to capture message-passing costs. An instance of the TAUMeasurement component provided timer support and the Mastermind component was created to record performance measurements for different execution combinations.

The simulation was run on three processors of a cluster of dual 2.8 GHz Pentium Xeons with 512 KB caches. `gcc` version 3.2 was used for compiling with `-O2` optimization. Performance timings revealed about 25% of the time is spent in `MPI_Waitsome()`, which is invoked from two methods in AMRMesh—one that does 'ghost-cell updates' on patches (gets data from abutting, but off-processor

patches onto a patch) and the other that results in load-balancing and domain (re-)decomposition. The other methods, one in States and the other in GodunovFlux, are modeled below.

In Figure 6(a) we plot the execution times for States for both the sequential and strided mode of operation. We see that for small, largely cache-resident arrays, both the modes take roughly the same time. As the arrays overflow the cache, the strided mode becomes more expensive and one sees a localization of timings. In Figure 6(b), we plot the ratio of strided and sequential access times. The ratio varies ranges from one for small arrays to around four for large arrays. Further, for larger arrays, one observes large scatters. Similar phenomena are also observed for both GodunovFlux and EFMFlux.

During the execution of the application, both the X- and Y-derivatives are calculated and the two modes of operation of these components are invoked in an alternating fashion. Thus, for performance modeling purposes, we consider an average. However, we also include a standard deviation in our analysis to track the variability introduced by the cache. It is expected that both the mean and the standard deviation will be sensitive to the cache size. In Figures 6(c), (d), and (e), we plot the execution times for the States, GodunovFlux, and EFMFlux components, respectively. Regression analysis was used to fit simple polynomial and power laws, which are also plotted in the figures. The mean execution time scales linearly with the array size, once the cache effects have been averaged out. The standard deviations exhibit some variability, but they are only significant for GodunovFlux, a component that involves an internal iterative solution for every element of the data array. Note that these timings do not include the cost of the work done in the proxies, since all the extraction and recording of parameters is done outside the timers and counters that actually measure the performance of a component.

If T_{States}, T_{Godunov} and T_{EFM} are the execution times (in microseconds) for States, GodunovFlux and EFMFlux and Q is the input array size, the best-fit execution time models for the three components are

$$
\begin{aligned}
T_{\text{States}} &= \exp(1.19 \log(Q) - 3.68) \\
T_{\text{Godunov}} &= -963 + 0.315Q \\
T_{\text{EFM}} &= -8.13 + 0.16Q
\end{aligned}
\tag{1}
$$

The corresponding expressions for the standard deviations σ are

$$
\begin{aligned}
\sigma_{\text{States}} &= \exp(1.29 \log Q) \\
\sigma_{\text{Godunov}} &= -526 + 0.152Q \\
\sigma_{\text{EFM}} &= 66.7 - 0.015Q + 9.24 \times 10^{-7} Q^2 - 1.12 \times 10^{-11} Q^3 + 3.85 \times 10^{-17} Q^4
\end{aligned}
\tag{2}
$$

We see that GodunovFlux is more expensive that EFMFlux, especially for large arrays. Further, the variability in timings for GodunovFlux increase with Q while it decreases for EFMFlux. While GodunovFlux is the preferred choice for scientists (it is more accurate), from a performance point of view EFMFlux has better characteristics. This is an excellent example of a quality of service issue where numerical and/or algorithmic characteristics (such as accuracy, stability, and robustness) may need to be added to the performance model. Thus, the performance of a component implementation would be viewed with respect to the size of the problem as well as the quality of the solution produced by it.

In Figure 6(f) we plot the communication time spent at different levels of the lowercase grid hierarchy during each communication ('ghost-cell update') step. We plot data for processor 0 first. During the course of the simulation, the application was load-balanced, resulting in a different domain decomposition. This is seen in a clustering of message-passing times at levels 0 and 2. Ideally, these
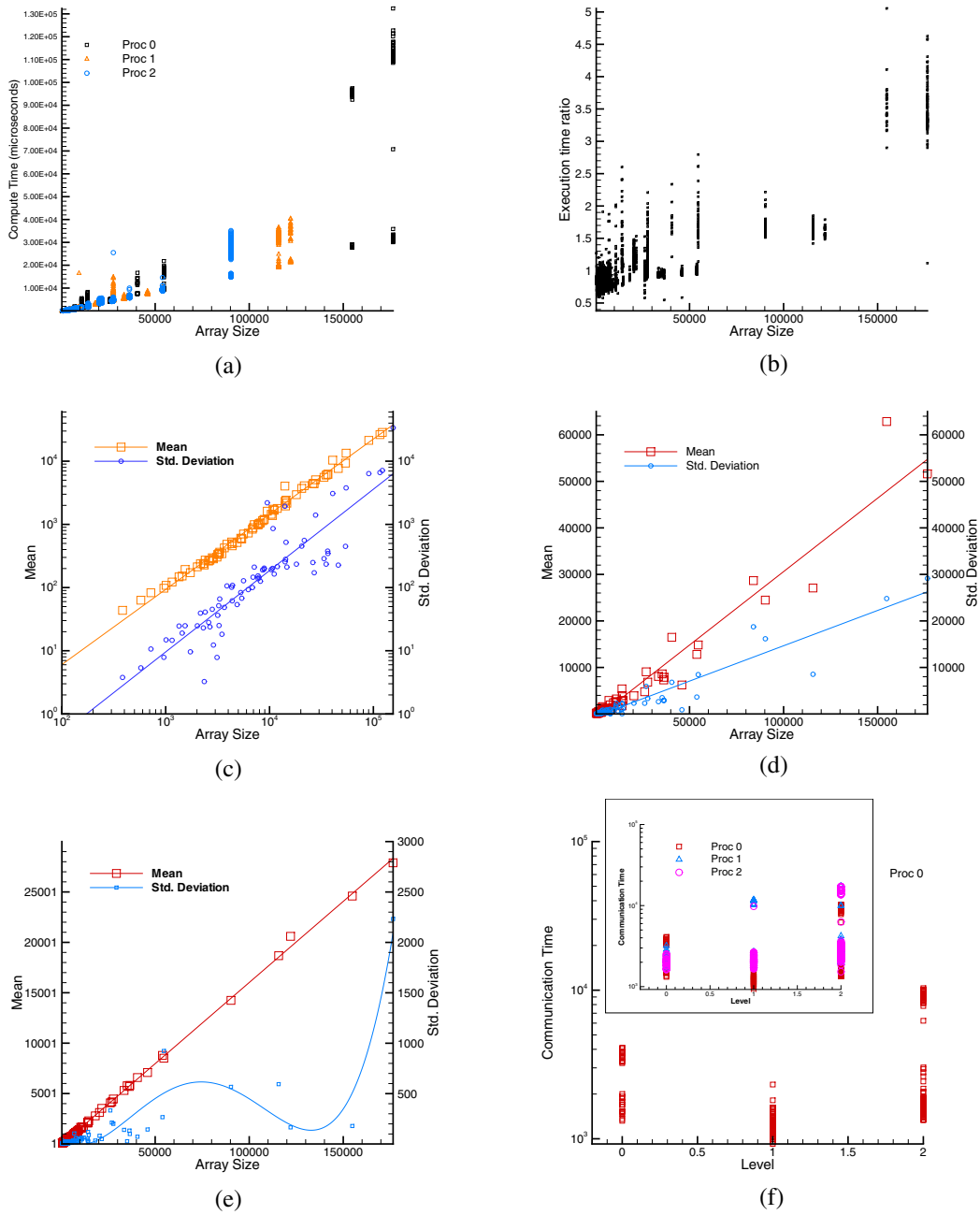
Figure 6. Performance measurements for shock application.

clusters should have collapsed to a single point; the substantial scatter is caused by fluctuating network loads. In the inset we plot results for all three processors. A similar scatter of data points is seen. Comparing with Figures 6(c), (d), and (e), we see that message-passing times are generally comparable to the purely computational loads of States and GodunovFlux, and it is unlikely that the code, in the current configuration (the given problem and the level of accuracy desired), will scale well. This is also borne out by performance profiles where almost a quarter of the time is shown to be spent in message passing.

The results of this example confirm the advantages of an integrated methodology and infrastructure for performance engineering in component software.

6. CONCLUSION

To leverage the power of software abstraction while maintaining high-performing applications demands a tight integration of performance measurement and analysis technology in the software engineering process. The success of component software for scientific applications running on large-scale parallel computers will be determined by how close performance comes to standard implementation approaches. We have designed a performance interface for component software, and implemented a performance component using the TAU performance system, to help understand performance problems in scientific component applications.

However, this is only one part of the story. Component software has the inherent abstractional power over standard approaches to modify what and how components are used in a computation. If it is possible to inform these component choices with performance information in a manner compatible with component software engineering design, there is great potential for adapting component applications to optimize performance behavior. Indeed, our demonstration shows how the TAU performance component can be used within CCA applications in ways that would allow high-performing, self-optimizing component solutions to be achieved.

However, to do so, we must better address the problems of performance modeling. The ultimate aim of performance modeling is to be able to compose a composite performance model and optimize a component assembly. Apart from performance models, this requires multiple implementations of a functionality (so that one may have alternatives to choose from) and a call trace from which the inter-component interaction may be derived. The wiring diagram (available from the framework) along with the call trace (detected and recorded by the performance infrastructure) can be used online to create a composite performance model where the variables are the individual performance models of the components themselves. This facilitates dynamic performance optimization, which uses online performance monitoring to determine when performance expectations are not being met and new model-guided decisions of component use need to take place.

While our work has focussed on HPC, there is a direct extension of the methods and infrastructure to Grid computing. Component software is a natural model for developing applications for the Grid, as evidenced by the ICENI [19,20] and CCAT [27] projects. Indeed, the CCA efforts include Grid-based implementations, such as XCAT [27,28], that use Globus [29] for security and remote task creation, and RMI over SOAP for communication. Because our approach leverages the abstraction power of CCA, as well as the infrastructure of CCA frameworks, we expect to similarly leverage Grid infrastructure and services through interfaces to technology such as XCAT. Our monitoring services can also benefit from the Grid monitoring architecture activities.

REFERENCES

1. Allan B, Armstrong R, Wolfe A, Ray J, Bernholdt D, Kohl J. The CCA core specifications in a distributed memory SPMD framework. *Concurrency: Practice and Experience* 2002; **14**:323–345.
2. Bramley R *et al.* Component architectures for distributed scientific problem solving. *IEEE Computational Science and Engineering* 1998; **5**(2):50–63.
3. Epperly T, Kohn S, Kumfert G. Component technology for high-performance scientific simulation software. *Federation for Information Processing Working Conference on Software Architectures for Scientific Computing*. Kluwer: Dordrecht, 2000.
4. Common Component Architecture Forum. http://www.cca-forum.org [19 November 2004].
5. Malony A, Shende S. Performance technology for complex parallel and distributed systems. *Distributed and Parallel Systems from Instruction Parallelism to Cluster Computing*, Kotsis G, Kacsuk P (eds.). Kluwer: Dordercht, 2000; 37–46.
6. Cleary A, Kohn S, Smith S, Smolinski B. Language interoperability mechanisms for high-performance scientific applications. *SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, October 1998. SIAM: Philadelphia, PA, 1998.
7. Kumfert G *et al.* Achieving language interoperability with Babel. CASC/ISCR workshop on object-oriented and component technology for scientific computing. *Technical Report UCRL-PRES-144649*, LLNL, 2001.
8. Szyperski C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley: Reading, MA, 1997.
9. Object Management Group. CORBA components. *OMG TC Document orbos/99-02-95*, 1999. Available at: http://www.omg.org.
10. Maloney J. *Distributed COM Application Development Using Visual C++ 6.0*. Prentice-Hall: Englewood Cliffs, NJ, 1999.
11. Englander R, Loukides M. *Developing Java Beans (Java Series)*. O'Reilly and Associates, 1997. Available at: http://www.java.sun.com/products/javabeans.
12. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes L, Parker S, Smolinski B. Toward a common component architecture for high-performance scientific computing. *High Performance Distributed Computing Symposium*. IEEE Computer Society Press: Washington, DC, 1999.
13. CCTTSS, DOE SciDAC Center for Component Technology for Terascale Simulation Software. http://www.cca-forum.org/scidac [19 November 2004].
14. Mniszewski S, Beckman P, Fasel P, Humphrey W. Efficient coupling of parallel applications using PAWS. *IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, 1998. IEEE Computer Society Press: Washington, DC, 1998.
15. St Germain J, McCorquodale J, Parker S, Johnson C. Uintah: A massively parallel problem solving environment. *High Performance Distributed Computing Conference*. IEEE Computer Society Press: Washington, DC, 2000; 33–41.
16. Mos A, Murphy J. Performance monitoring of Java component-oriented distributed applications. *IEEE International Conference on Software, Telecommunications and Computer Networks—SoftCOM*. IEEE Press: Piscataway, NJ, 2001.
17. Sridharan B, Dasarathy B, Mathur A. On building non-intrusive performance instrumentation blocks for CORBA-based distributed systems. *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium*. IEEE Press: Piscataway, NJ, 2000.
18. Sridharan B, Mundkur S, Mathur A. Non-intrusive testing, monitoring and control of distributed CORBA objects. *TOOLS Europe 2000*, June 2000. IEEE Computer Society Press: Washington, DC, 2000.
19. Furmento N, Mayer A, McGough S, Newhouse S, Field T, Darlington J. Optimisation of component-based applications within a Grid environment. *Supercomputing*. ACM Press: New York, 2001.
20. Furmento N, Mayer A, McGough S, Newhouse S, Field T, Darlington J. ICENI: Optimisation of component applications within a Grid environment. *Parallel Computing* 2002; **28**:1753–1772.
21. Shende S, Malony A, Rasmussen C, Sottile M. A performance interface for component-based applications. *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems, IPDPS'03*. IEEE Computer Society Press: Washington, DC, 2003.
22. Bell R, Malony A, Shende S. A portable, extensible, and scalable tool for parallel performance profile analysis. *Euro-Par 2003* (*Lecture Notes in Computer Science*, vol. 2790). Springer: Berlin, 2003; 17–26.
23. Nagel W, Arnold A, Weber M, Hoppe HC, Solchenbach K. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer* 1996; **12**(1):69–80.

24. Lindlan K, Cuny J, Malony A, Shende S, Mohr B, Rivenburgh R, Rasmussen C. A tool framework for static and dynamic analysis of object-oriented software with templates. *Supercomputing*. IEEE Computer Society Press: Washington, DC, 2000.
25. Berger M, Collela P. Local adaptive mesh refinement for shock hydrodynamics. *Journal Computational Physics* 1989; **82**:64–84.
26. Quirk J. A parallel adaptive Grid algorithm for shock hydrodynamics. *Applied Numerical Mathematics* 1996; **20**(4):427–453.
27. Villacis J *et al.* CAT: A high performance, distributed component architecture toolkit for the Grid. *High Performance Distributed Computing Conference*. IEEE Computer Society Press: Washington, DC, 1999.
28. XCAT Project. http://www.extreme.indiana.edu/xcat/ [19 November 2004].
29. Globus. http://www.globus.org [19 November 2004].