Computational Quality of Service for Scientific CCA Applications: Composition, Substitution, and Reconfiguration

Lois Curfman McInnes,[1] Jaideep Ray,[2] Rob Armstrong,[3] Tamara L. Dahlgren,[4]
Allen Malony,[5] Boyana Norris,[1] Sameer Shende,[5] Joseph P. Kenny[3], and Johan Steensland[2]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 USA
{mcinnes,norris}@mcs.anl.gov [‡]
[2] Advanced Software R&D, Sandia National Laboratories, Livermore, CA 94551 USA
{jairay,jsteens}@ca.sandia.gov
[3] Scalable Computing R&D, Sandia National Laboratories, Livermore, CA 94551 USA
{rob,jpkenny}@ca.sandia.gov [§]
[4] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, 94551 USA
dahlgren1@llnl.gov [¶]
[5] Computer Science Department, University of Oregon, Eugene, OR 97403 USA
{malony,sameer}@cs.uoregon.edu [‖]

February 2006

Abstract. Component-based design can help manage the complexity of high-performance scientific simulations, where it has become increasingly clear that no single research group can effectively develop, select, or tune all of the components in a given application and that no single tool, solver, or solution strategy can seamlessly span the entire spectrum efficiently. Component approaches augment the benefits of object-oriented design with programming language interoperability, common interfaces, and dynamic composability. Our work addresses the challenge of how to compose, substitute, and reconfigure components dynamically during the execution of a scientific application. The goal is to make suitable compromises among performance, accuracy, mathematical consistency, and reliability when choosing among available component implementations and parameters. As motivated by high-performance simulations in combustion, quantum chemistry, and accelerator modeling, this paper discusses ideas on computational quality of service (CQoS) — the automatic selection and configuration of components to suit a particular computational purpose. We discuss the synergy between component-based software design and CQoS, with emphasis on features of the Common Component Architecture that provide the foundation for this work. We introduce the design of our CQoS software, which consists of tools for measurement, analysis, and control infrastructure, and we discuss directions of future work.

1 Introduction

As computational science progresses toward ever more realistic multiphysics and multiscale applications, the complexity is becoming such that no single research group can effectively develop, select, or tune all of the components in a given application, and no single tool, solver, or solution strategy can seamlessly span the entire spectrum *efficiently*. A goal of component technology is to help manage this complexity by augmenting the benefits of object-oriented design with programming language interoperability, dynamic composability, and common interfaces for particular functionalities [65]. Interchangeable components based on varying characteristics such as their underlying models, precision, space requirements, execution performance, and reliability were key features of the vision published in McIlroy's 1968

‡ Research at Argonne National Laboratory was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract W-31-109-ENG-38.

§ Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U.S. Dept. of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

¶ Some of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

‖ Research at the University of Oregon is sponsored by contracts (DEFG03-01ER25501 and DE-FG02-03ER25561) from the MICS program of the U.S. Dept. of Energy, Office of Science.

seminal paper on software components [48]. Common component interfaces enable easy access to suites of independently developed algorithms and implementations, and dynamic composability facilitates switching among different implementations during runtime. The challenge then becomes how to automatically make sound choices from among the available implementations and parameters, with suitable tradeoffs among performance, accuracy, mathematical consistency, and reliability. Such choices are important both for the initial composition and configuration of an application and for adaptive control during runtime.

We are addressing this challenge by developing tools for *computational quality of service* (CQoS) [52], or the automatic selection and configuration of components to suit a particular computational purpose. CQoS embodies the familiar concept of quality of service (QoS) in networking as well as the ability to specify and manage characteristics of the application in a way that adapts to the changing computational environment. QoS research issues for scientific component software differ in important ways from more common QoS approaches that often emphasize system-related performance effects such as CPU or network loads to implement application priority or bandwidth reservation in networking. Although performance is a shared general concern, high efficiency and parallel scalability are more significant requirements for scientific components, along with *functional* qualities, such as the level of accuracy achieved for a particular algorithm.

The remainder of this paper is organized as follows: motivation, component-based design, related work, CQoS architecture, preliminary investigations, and conclusions. More specifically, Section 2 introduces three high-performance scientific applications that motivate this work: parallel partitioning of meshes in combustion simulations, evaluation of molecular wave functions in quantum chemistry models, and the solution of linear systems arising in high-energy accelerator simulations. Section 3 discusses the synergy between component-based software design and CQoS and provides an overview of the Common Component Architecture (CCA) [4, 6, 15] — a component model that provides the foundation for this work and has been specifically designed for high-performance scientific computing. Section 4 describes related work. Section 5 introduces the design of our CQoS software infrastructure, which consists of two main groups of tools: (1) measurement and analysis infrastructure and (2) control infrastructure. Our preliminary investigations into adaptive strategies, performance analysis, and basic interface semantics are described in Section 6. Finally, Section 7 discusses conclusions and directions of future work.

2 Motivating Scientific Applications

Computational quality of service (CQoS) can generally be interpreted as the ability of a system to solve a scientific problem with the best available hardware and software tools. In scientific computing, "best" has typically meant robust and fast; the tools are expected to solve problems of varying degrees of difficulty consistently, reliably, and efficiently. However, because scientific problems have become increasingly complex, it is impossible to design a single, efficient solution strategy that seamlessly spans the entire spectrum of problems.

Hence, the following techniques are primarily used for improving the CQoS of scientific computations: *adaptation* and *modularization*. Adaptation typically exploits temporal and spatial problem-specific features to concentrate computational resources in critical regions. Although adaptation can reduce the computational resources needed for a given problem to a fraction of static techniques, adaptation introduces a high degree of unpredictable dynamics. Modularization exploits segregation in coupling — loosely coupled subproblems are identified and solved by efficient, highly specialized software. Suitable software components for the given problem are typically selected and compiled before the program is executed, so that modularization is commonly thought of as being static.

Most efficient scientific applications currently use adaptation *or* modularization. A framework that allows the simultaneous use of both techniques is nonexistent as far as we know. But there is a previously unexploited symbiotic relation between the two. To exploit this symbiosis for improving efficiency and, hence, the CQoS for dynamic scientific problems, we introduce *method adaptation*: During decision points of an executing application, the most suitable solution technique is dynamically selected and configured based on the current state of the problem. Solution techniques can easily be implemented as software modules (or components), thereby enabling method adaptation through *dynamic reconfiguration* of the code. This can be accomplished either by dynamically setting configurable parameters or replacing modules with more appropriate ones. Replacement modules must exist in a component repository and do not necessarily need to be instantiated during the beginning of a given simulation. Such an on-demand strategy dispenses with the need to collate all available software into a single huge and difficult-to-maintain library. Furthermore, this piecemeal approach to adaptation is scalable in terms of the number and variety of modules that can be accommodated in such a manner.

Previous work [10,55,64,67] suggests that it is indeed possible to develop a divide-and-conquer approach that uses CQoS for composition, substitution, and reconfiguration of components within long-running simulations. In Section 5 we introduce ideas for such software infrastructure, where our approach leverages related work (e.g., [28,44]) when appropriate. We suggest how to apply these tools to three scientific simulations, the details of which are very different. Part of our research is to identify the common as well as the application-specific aspects of these problems as a first step toward designing a strategy that will make CQoS-enabled simulations of the three problems a reality.

Three parallel scientific applications that motivate this research represent different computational science disciplines and involve either partitioning, resource management, or linear solvers. This section describes each and mentions our associated plans.

Partitioning meshes in combustion simulations. The Computational Facility for Reacting Flow Science [53], funded by the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) initiative [68], is developing a CCA toolkit for simulating flames on block-structured adaptive meshes. For large and realistic simulations executed on parallel computers, the mesh is partitioned and distributed across processors. Unfortunately, no single partitioning algorithm is suitable for all computer and application states [62]. A meta-partitioner [64] selects and configures the most suitable partitioner based on system and application state. In this context, the dynamic selection of partitioning algorithms for improving scalability corresponds to method adaptation. Instant mesh characterization [63, 64] and rigorous partitioner characterization with respect to partitioner parameters [37] allow for a mapping between application state and partitioner configuration. Since the adaptive mesh changes as the simulation evolves, this mapping is performed repeatedly. To implement the meta-partitioner, we envision encapsulating each partitioner in a software component and then dynamically switching these components automatically based on the application state.

Resource management in quantum chemistry simulations. Quantum chemical computations typically involve multiple subproblems [41], each of which places significant demands on system resources. The predominant methods for molecular wave function determination require the computation of large numbers of integrals over atom-centered basis functions, followed by the optimization of coefficients for forming molecular orbitals from these atom-centered functions. A typical resource competition arises in this case, as memory must be partitioned between the atomic integral computation, where the storage of intermediate quantities reduces computational effort, and the molecular orbital determination, where the storage of previous coefficient vectors speeds the convergence of iterative procedures. To date, resource distribution has been statically defined by the user prior to execution of the task, requiring an experienced user who can make accurate estimates of resource demands and relative performance costs. Clearly, the consequences of reliance on the end-user for resource distribution are lower computational efficiency and lost processor cycles. The addition of dynamic resource adaptation to our existing component modules will regain lost system time and enable the automation of large numbers of tasks, as required in complex multiscale simulations, while maintaining high computational efficiency.

Solving linear systems in high-energy accelerator simulations. Solving linear algebraic systems of equations often dominates the overall execution time of large-scale simulations based on partial differential equations (PDEs), including some facets of accelerator modeling. In this context, CQoS focuses on selecting and configuring parallel linear solver components [60] under development by the Terascale Optimal PDE Simulations (TOPS) project [17], based on the context of the overall simulation and the properties of the coefficient matrix that defines the linear system. Because the properties of linear systems in time-dependent and/or nonlinear applications may significantly change during the course of a given simulation, CQoS-enabled adaptive multimethod solvers have promise to improve robustness and reduce overall time to solution [10, 11, 50]. Our approach will leverage related work by Eijkhout and Fuentes [28] on matrix characterization and metadata and by Bhowmick et al. [8] on machine learning.

Because these problems appear to have little in common, we expect the logic involved in characterizing them and choosing efficient solution strategies to be vastly different. However, we believe that the *infrastructure* for analyzing and characterizing each problem (the mesh, molecular wavefunction computation, and linear system) and determining and invoking solution strategies will indeed be similar. It is this (conjectured) separation of *logic* and *infrastructure* that we seek to verify in this work and, if verified, exploit to enable CQoS in these very different problems.

3 CQoS and Component-Based Software Design

Complexity generally succumbs to modularity; being able to identify loosely coupled subproblems allows the focusing of targeted strategies to a particular problem instance. Sometimes, individual strategies may be combined into general ones whose behavior may be made to vary continuously in a parametric fashion. This realization has led to *libraries* that embody various tools and algorithms to address a particular class of problems, with each being appropriate for problems of a given type. However, the library approach is quickly approaching its limits of applicability for two reasons:

1. Maintaining a collection of algorithms and tools under a single roof poses a daunting challenge and is not a scalable approach if the bulk of contributors are transient collaborators.
2. The choice of algorithm/tool to use for a given problem instance is typically left to the user (often, not an expert in the field), who invariably chooses the most reliable, if inefficient, approach.

Each problem can place a significant hurdle to addressing the kind of scientific questions that ultrascale computing can enable.

The first problem has a conceptually simple solution: instead of a monolithic approach, one maintains a "stable" of algorithms, individually implemented as independent or peer components. Component-based software architectures can reliably enable such an approach. Further, many component-based architectures allow adaptive or dynamic recomposition of codes during runtime so that components best suited to the problem at hand can be loaded to replace or reconfigure the current solution infrastructure. This feature simply and elegantly solves the problem of *bringing* in the tools best suited to the problem. *Choosing* the appropriate tools (or an appropriate set of parameters to configure a general tool) thus becomes the main hurdle to fashioning a dynamic and adaptive strategy. The choice will typically be made predicated on accuracy, stability, efficiency, or performance —— tangible metrics that can be melded into objectives for *computational quality of service* (CQoS). CQoS addresses the question of formulating and designing the *control system* that makes the choice, with suitable tradeoffs among performance, accuracy, mathematical consistency, and reliability.

The potential of CQoS is not without challenges. Parallel execution affects all criteria in component-dependent ways. Understanding the relationships is difficult, as is representing this knowledge in some form. Dynamic CQoS implies a dynamic awareness of computational state and execution history. Runtime observation must be implemented in some manner and necessarily results in (nonfunctional) overhead in the computation, imposing a computational performance tradeoff that may or may not be important depending on objectives. These problems are difficult enough from the perspective of a single component. Understanding component compositions and full multicomponent applications is significantly more complex. The optimization problem is intractable, in general. However, the approach of component-based software development makes CQoS conceivable. That is, the framework used for component development, composition, application construction, and execution provides an architecture for CQoS engineering. Thus, we can design tools that support CQoS implementation, that are consistent with the component software methodology and may even be implemented by using component technology. In the following, we discuss goals and approaches for such CQoS tools that we are developing.

The Common Component Architecture. Our work employs the Common Component Architecture (CCA) [4,6,15], which has been designed specifically for the needs of parallel, scientific high-performance computing in response to limitations in this domain of other, more widely used component approaches. A comprehensive description of the CCA, including a discussion of how it differs from other component models, is available [6]; here we present a brief overview of the CCA environment, focusing on the aspects most relevant to CQoS infrastructure.

The specification of the Common Component Architecture [14] defines the rights, responsibilities, and relationships among the various elements of the model. Briefly, the elements of the CCA model are as follows:

- *Components* are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces.
- *Ports* are the abstract interfaces through which components interact. Specifically, CCA ports provide procedural interfaces that can be thought of as a class or an interface in object-oriented languages, or a collection of subroutines, or a module in a language such as Fortran 90. Components may provide ports, meaning that they implement the functionality expressed in a port (called *provides* ports), or they may use ports, meaning that they make calls on a port provided by another component (called *uses* ports).

– *Frameworks* manage CCA components as they are assembled into applications and executed. The framework is responsible for connecting *uses* and *provides* ports without exposing the components' implementation details. The framework also provides a small set of standard services that are available to all components. Several frameworks that implement the CCA specification and support various computing environments have been developed. Ccaffeine [1] is used by the applications discussed in Section 2.

The CCA's general port mechanism, along with various specific ports, make it possible for us to address these CQoS issues. In particular, CCA-defined service ports, such as `ConnectionEventService`, `BuilderService`, and `AbstractFramework`, are required of all frameworks. `ConnectionEventService` notifies components when connections are made and broken. `BuilderService` and `AbstractFramework` provide a means to programmatically assemble and modify applications (instantiate and destroy components, make and break connections between ports) and a means for arbitrary code to become a CCA framework. These services allow dynamic monitoring and control of component applications by CQoS infrastructure, for example, enabling the implementation of control components that swap application components based on CQoS control laws [33, 52].

4 Related Work

Adaptive software for scientific computing is clearly an area of emerging research, as evidenced by a number of recent projects and related work [13, 16, 22–25, 27, 28, 32, 38, 39, 43, 44, 47, 57, 59, 61, 66, 69–73, 77]. In general, support for assembly, substitution, and reconfiguration requires the identification of relevant characteristics of components and assessment of application behavior at runtime. The former necessitates the availability of higher-level semantic information that is machine processable; the latter requires some form of runtime monitoring. This section summarizes a number of efforts from the literature that address one or both of these issues.

Three approaches of interest for specifying semantic information are models, contracts, and service-level agreements. Furmento et al. [30] as well as Gu and Nahrstedt [31] discuss performance models and their use in overall component application assembly at runtime within the context of distributed environments; Beugnard et al. [7] define a general model of software contracts and discuss approaches for making components contract-aware. Similarly, the SAMcode model of adaptable mobile agents [2] allows the specification of contracts — consisting of one precondition and one postcondition — for each adaptable method. Violations are used to select between different implementations of a method at runtime. The GlueQoS work of Wohlstadter et al. [76] focuses on mediating quality-of-service requirements — specified as assertions — between clients and Web services. Bennett et al. [5] discuss the need for service-level agreements for defining the terms and conditions of use, with agreements providing a minimum of coupling between components. They also emphasize the importance of characterizing relevant component features to ensure both the correct use and provision of services. Raje et al. [54] describe a QoS framework for distributed, heterogeneous components and provide a catalog of QoS metrics [12]. The Software-Implemented Fault Tolerance (SIFT) environment for Adaptive Reconfigurable Mobile Objects of Recovery (ARMOR) processes [74] relies on their model for functional reconfiguration to adjust application behavior to meet dependability requirements. In this case adaptation is accomplished through user-specified assertion checks at critical execution points and the use of microcheckpointing to adjust application state accordingly. In addition, Loyall et al. [45] use semantic information in distributed object systems. Hence, the usefulness of models, contracts, and service-level agreements mechanisms for defining component and application semantics has been demonstrated in a variety of contexts.

Relevant techniques for runtime behavioral monitoring can be directly or indirectly dependent upon events. Reiner and Pinkerton [56] explore dynamically changing control parameters to improve operating system performance and use experiments to determine improved settings. They develop a methodology for adaptive tuning as well as algorithm, policy, and (fixed) parameter selection. Whisnant et al. [74] rely on human intervention to deal with reconfiguration after a problem is detected at runtime. Feather et al. [29], however, use event monitoring of behavioral deviations and changing environmental conditions to reconcile the intended system behavior with individual requirements at runtime. In these cases, monitoring an application at runtime involves checking control parameters and monitoring events, including application failure.

Unlike these efforts, our approach relies on high-level interface specifications and technologies tailored for scientific computing. Quality-performance tradeoffs for parallel scientific simulations will be made using the Common Component Architecture (CCA). Because the glue that binds CCA components together is a set of common, agreed-upon interfaces, multiple component implementations conforming to the same external interface standard are interoperable. The common interface specifications provide the flexibility to accommodate different algorithms, performance characteristics, and coding styles in multiple implementations.

5 CQoS Software Architecture

This section describes the design of the CQoS software infrastructure and examines implementation approaches for the principal subsystems. Figure 1 illustrates our vision of how CQoS infrastructure will help to analyze, select, and parameterize components for the motivating applications introduced in Section 2. This diagram shows the two main facets of our CQoS tools: (1) *measurement and analysis infrastructure*, which combines performance information and models from historical and runtime databases along with interactive analysis, including statistical analysis and machine learning technology (further discussed in Section 5.1); and (2) *control infrastructure*, which encompasses decision-making components that evaluate progress based on domain-specific heuristics and metrics, along with services for dynamic component replacement (further discussed in Section 5.2). These two groups of CQoS tools, which may be employed both for initially composing an application and for runtime control, are largely decoupled and interact primarily through a substitution assertion database. Preliminary research that has led to this approach is discussed in [33,46,52,55,67].

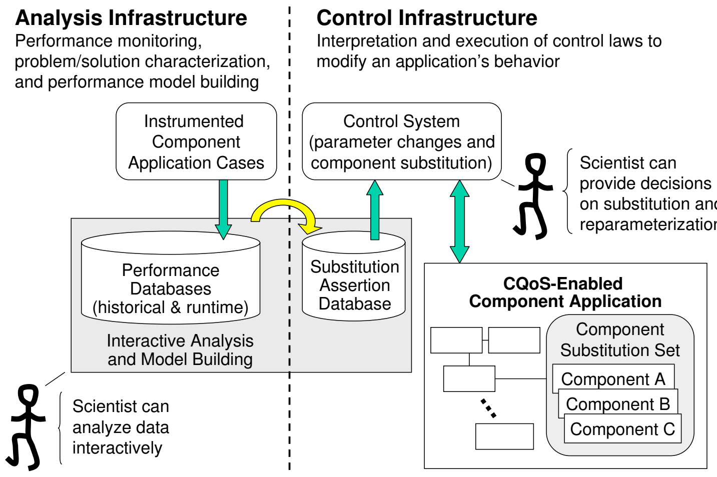
Analysis Infrastructure
Performance monitoring, problem/solution characterization, and performance model building
Control Infrastructure
Interpretation and execution of control laws to modify an application's behavior
Instrumented Component Application Cases
Control System (parameter changes and component substitution)
Scientist can provide decisions on substitution and reparameterization
Performance Databases (historical & runtime)
Substitution Assertion Database
Interactive Analysis and Model Building
CQoS-Enabled Component Application
Component Substitution Set
Component A
Component B
Component C
Scientist can analyze data interactively

Fig. 1. Overview of CQoS infrastructure.

At the very outset, we realize that a CQoS-enabling control system for high-performance scientific computing must adhere to two constraints:

1. Enabling CQoS in a given component-based scientific application cannot require extensive changes to the code. In fact, anything more than trivial changes may render the effort unpalatable to most application teams. Many scientific components have *sensors*, variables that characterize the current performance of a component (for example, the number of iterations taken to solve a linear system given a desired error) and *actuators*, configuration parameters that can change the behavior of a component (for example, a preconditioner that the solver might use). Enabling CQoS might, at most, require that such sensors and actuators be exposed in a CQoS-blessed way. Note that this terminology is borrowed from [44].
2. CQoS has to be enabled in a strictly additive manner; that is, a given component assembly comprising a scientific simulation may be augmented with a CQoS-enabling component to bring the advantages of CQoS to bear. If a CQoS-enabled scientific simulation is shorn of its CQoS-enabling components, however, the code should still function and provide correct results, if at a lower efficiency.

In addition, our CQoS system will not impose restrictions on the types of tools used for instrumentation or measurement, algorithms for decision making, or mechanisms for switching between components. We will define the flow of events when CQoS ports are connected, the mechanisms for communicating the need to run an optimizer at a given instance, and the way the decisions made by an optimizer are interpreted by those components that are connected to its CQoS port. In short, our CCA effort will assist in defining the protocol for interactions with the CQoS substrate and ways to automate the creation of CQoS-aware component ensembles.

5.1 Measurement and Analysis Infrastructure

As shown in Figure 2, infrastructure for measurement and analysis supports the collection of performance data and subsequent processing through statistical analysis and machine learning techniques for the purpose of creating performance models for key components. The performance models are used to develop application-specific control laws, which are stashed in a substitution assertion database and then employed by the complementary CQoS control infrastructure, as discussed in Section 5.2.

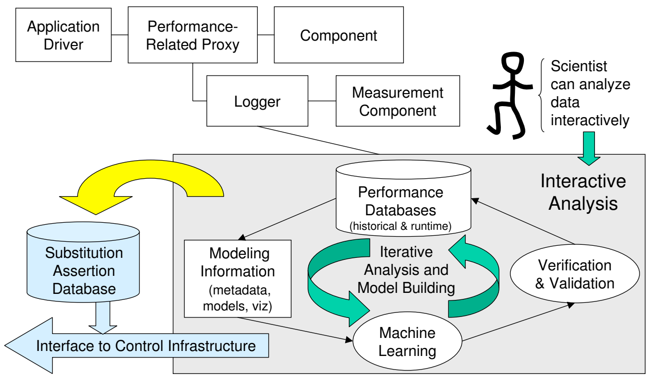
Application Driver
Performance-Related Proxy
Component
Logger
Measurement Component
Scientist can analyze data interactively
Interactive Analysis
Performance Databases (historical & runtime)
Substitution Assertion Database
Modeling Information (metadata, models, viz)
Iterative Analysis and Model Building
Verification & Validation
Machine Learning
Interface to Control Infrastructure

Fig. 2. CQoS analysis infrastructure, which interfaces to the complementary CQoS control infrastructure primarily via a substitution assertion database.

Performance measurement. The input to the analysis consists of performance measurements, along with application-specific problem characteristics and metrics. This data will be collected by using the TAU [58] performance system to generate performance monitoring proxies for components, taking advantage of fully automatic instrumentation at the component interface level, at the message-passing level, and within each component. In addition, application-specific metrics can be monitored by TAU's user event mechanism. This holistic view of performance data will facilitate assessment of an application's performance characteristics at any point during execution.

A performance database will store performance data gathered from prior executions, or trials, of the application. The metadata for a trial will include fields describing the particular machine, compilation parameters, application-specific runtime parameters, and any other relevant parameters that describe a unique application instance. We will use TAU's PerfDMF [34] database format to store and access performance data. A performance knowledge component will provide interfaces for querying historical performance data.

Performance analysis. The analysis portion of the CQoS infrastructure will use the performance database to discover significant performance features of applications by using statistical analysis and machine learning tools. We will employ PerfExplorer [35], which provides a single common interface to different general-purpose tools, including the R system [36], WEKA [75], and Octave [26], and will be extended with more tools in the future.

Performance models. Generating component modeling information is an important part of the CQoS effort. The objective is to translate information about a component's computational properties, as may be captured in analytical models and empirical observations obtained from parametric experimentation, into predictive CQoS models that can be used to evaluate CQoS expectations (assertions on current CQoS state) and direct control decision (assertions on future CQoS state).

1. In an *analytical approach*, metrics describing a given type of problem are computed for a particular instance. For example, this approach may be used in the selection of partitioners; an incoming mesh (actually, a block-structured adaptive mesh) is evaluated for its "partitionability" for load-balancing and communication, and a partitioner is selected and configured accordingly [37, 63, 64]. Similar work is under way for the characterization of linear solvers [8].
2. In an *empirical approach*, the performance of various components is continuously monitored and checked against an "accepted" norm. This approach is employed when various implementations are equally acceptable (from mathematical and scientific points of view), and the choice is made based on performance, such as the suitability of a particular algorithm to the problem at hand (which in most simulations evolves in time), the suitability of an implementation of the algorithm to the machine architecture, and so forth. In such cases, one has to define a baseline performance and codify it in a performance model, which is then used as the norm. Any significant deviations from the norm trigger the control law and a modification of behavior [55]. This work builds on our work in performance data mining, implemented by the PerfExplorer framework, which supports cluster, correlation, and comparative analysis techniques.

The knowledge represented in the CQoS models, as derived from the analytical and empirical analysis, should be as robust as possible to provide good predictive power but also as compact as possible to be quickly evaluated during execution. The CQoS models will be stored in the substitution assertion database for later use as part of an application-specific control law, which is further discussed in Section 5.2.

5.2 Control System

The role of the control subsystem of the CQoS architecture is to evaluate progress based on domain-specific heuristics and to provide mechanisms for dynamic component (re)configuration or replacement. CQoS addresses the question of formulating and designing the *control system* that delivers the required performance, while maintaining a suitable compromise among performance, accuracy, consistency, and robustness. We consider this as a sum of two loosely coupled parts:

1. a *control law* that characterizes the problem presented to it and chooses a suitable tool to deal with it, and
2. a *control infrastructure* that implements the dictates of the control law.

Control laws. Control laws are the core of the control system. Each control law, as developed during the performance analysis phase introduced in Section 5.1, is in charge of characterizing a problem, characterizing the elements it manages, and establishing a mapping between the two. We expect each control law to be tied (in the mathematical, not the software sense) to the elements (algorithms/tools) that it manages (e.g., the mesh partitioners, molecular wavefunction computations, and linear solvers introduced for the motivating scenarios in Section 2). Because we envision that the component codes will be dynamically reconfigurable, the elements that a control law manages will not generally be available for querying at runtime; thus, this information will have to be externally available. While this situation requires an abstract categorization of the elements being managed (e.g., a linear solver may be able to handle nonsymmetric systems with a moderate spectrum of condition numbers), it also enables the control law to exist as an independent entity in a substitution assertion database, as introduced in Section 5.1.

The control law's net output will be recommendations about appropriate tools to be used; implementing the recommendation is left to the control infrastructure. Such recommendations may range from merely modifying parameters in a currently instantiated tool/algorithm/component to outright replacement of components. In a parallel context, outright replacement of components will require a global view of the problem, thus incorporating elements of the parallel machine's configuration and characteristics in the decision process. Also, more generally, the control law will have to embody some degree of closed-loop feedback to maintain stability. Similar issues in recommender systems have been addressed by Houstis et al. [32].

Control infrastructure. Figure 3 depicts an overview of our approach for control infrastructure, which has the function of implementing the control law in component-based software. We view the control infrastructure as a collection of components "decorating" a component-based scientific simulation to CQoS-enable it. Two key parts are a *substitution and reparameterization decision service* and a *replacement service*. As introduced in [33], the dynamic replacement service arranges for the seamless replacement of one component implementation by another that provides the same

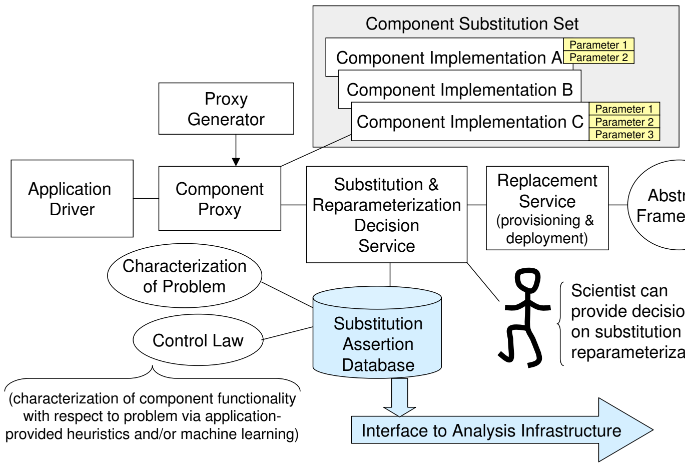
Component Substitution Set
Component Implementation A
Parameter 1
Parameter 2
Component Implementation B
Component Implementation C
Parameter 1
Parameter 2
Parameter 3
Proxy Generator
Application Driver
Component Proxy
Substitution & Reparameterization Decision Service
Replacement Service (provisioning & deployment)
Abstract Framework
Characterization of Problem
Control Law
Substitution Assertion Database
Scientist can provide decisions on substitution and reparameterization
(characterization of component functionality with respect to problem via application-provided heuristics and/or machine learning)
Interface to Analysis Infrastructure

Fig. 3. CQoS control infrastructure, which interfaces to the complementary CQoS analysis infrastructure primarily via a substitution assertion database that contains control laws.

functionality but has different performance features. The replacement service, which provisions and deploys the new component, can be invoked by proxy either on the programmer's behest or triggered by the substitution decision service. The substitution decision service automates decisions about component replacement by applying the control law, which resides in the substitution assertion database and may be supplied either by an application scientist or by a machine learning module. A similar approach has been used by [44].

We note that the substitution/replacement decision service itself runs in parallel in the application. Depending on the component's CQoS model, local and global management operations will be performed, for example, to collect and evaluate current CQoS state, determine global CQoS metrics, and reach a consensus decision on replacement. Figure 4 illustrates this situation.

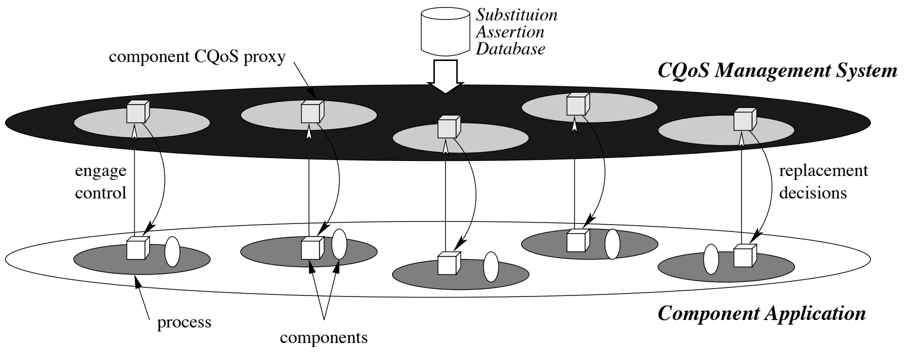

Fig. 4. CQoS infrastructure for replacement of components.

5.3 Application Interfaces

A central objective of the design and implementation of the CQoS infrastructure is to minimize the changes required for application components to take full advantage of CQoS capabilities for monitoring, analysis, and control. In this section we present an overview of our approach to achieving this objective.

Component proxy ports. As shown in Figures 2 and 3, proxy components can help in the design of CQoS-aware components. A proxy component provides a convenient mechanism to interpose performance instrumentation between a *uses* and a *provides* port connection between two components. The TAU component infrastructure can generate proxy components automatically and on the fly by parsing the interface definition or the source code of a component. By interposing a proxy between a caller (*uses* port) and a callee (*provides* port), not only can we measure the performance associated with a port, but we can also tap into a wealth of application-level information, such as method arguments that flow through the caller-callee edge of a component interconnection graph. Moreover, by examining the calling stack of instrumented components using TAU's callgraph profiling, we can deduce the dynamic connections between components.

Proxy components will be augmented with CQoS tracking and component reparameterization/substitution capabilities. When a CQoS port of a proxy is not connected, its role will be restricted to performance instrumentation and measurement. When a proxy's CQoS port is connected to an optimizer component, the proxy will generate events to activate the performance selection capabilities of the optimizer. By reading the choice of an appropriate port on a CQoS port, the proxy will be able to connect a caller's *uses* port to the chosen *provides* port. Thus, a proxy will act as a switch connecting a *uses* port to an appropriate *provides* port at runtime. This switching among components will need to be coordinated with a global service to ensure that it is carried out in all contexts of the component assembly.

In this manner, an application composed of CCA components will be CQoS enabled by using proxy components without any changes to the source code of the individual components. By building the CQoS infrastructure with reusable components and ports, we will also support the scenario where an application developer decides to build CQoS-aware intelligent components that do not rely on proxies but instead directly connect to a CQoS port of an optimizer component.

Issues in interface semantics. To make the vision of interchangeable components a reality for scientific software, we must take that which is implicit — or in textual documentation — and make it explicit in a concise, human-readable and machine-processable form. Hence, an important goal is to identify and support relevant component interface semantics for expressing characteristics and constraints needed for dynamic adaptation.

This work involves investigating the expressiveness and suitability of general-purpose interface semantics specification mechanisms for automated behavioral adaptation tools applied across disciplines within computational science. Therefore, we will investigate behavioral and quality-of-service specifications for issues such as support for algorithm characteristics and constraints, precision, and result quality. Stable attributes can include whether an algorithm is implicit or explicit, whether the storage order of arrays is exchanged through its interfaces, or whether a two-dimensional or three-dimensional space is represented. We expect stable characteristics such as these to be expressible in an implementation language-neutral form, such as the Scientific Interface Definition Language (SIDL) [18], while dynamic characteristics will require representation in a more flexible format.

We will extend the SIDL/Babel toolkit [18] with constructs to express the static information. Preliminary work has already enhanced the basic syntactic descriptions of the component's application programming interface (API) through the addition of nonnegotiable behavioral contracts [20,21]. We will similarly extend SIDL to support the specification and inheritance of quality metrics and requirements. For example, a general interface for linear system solution may be annotated with CQoS metrics for execution time and accuracy. An interface for a specific linear system solution approach, such as an iterative solver, would inherit the CQoS metrics from its parent and may define new ones, such as the number of iterations or desired accuracy. Another descendant of the general linear solver interface, for example an interface for direct solvers, would naturally not have the number of iterations but may need to specify the cost of reordering.

Work on interface semantics addresses the need to more explicitly define the behavior and quality of services in a concise, human-readable and machine-processable form. Integrating static information into SIDL and dynamic information into the control infrastructure will enable runtime adaptation according to a variety of behavioral and service quality criteria.

5.4 A CQoS Testbed

To motivate and validate the CQoS infrastructure, we will develop a *CQoS Testbed* with key components drawn from the motivating scenarios introduced in Section 2. Because the goal of CQoS infrastructure is to be as automatic as possible without taking away developers' ability to specify the analysis and control decisions, the abstractions that constitute the infrastructure must accommodate both automation and human intervention at every level. During early phases of this work, a CQoS component or application will likely rely heavily on specialized code, with increased automation being incorporated in later phases. We must define both pragmatic and generalizable interfaces for CQoS that meet these requirements. We thus will build a testbed of components taken directly from relevant scientific applications, or reasonable facsimiles of such components for the purpose of experimenting with interface abstractions for CQoS that will be mainstreamed into application codes.

6 Preliminary Investigations

We next discuss preliminary work that has led to this approach, including adaptive strategies, performance measurement and analysis, and software quality through contracts.

6.1 Adaptive Strategies

Preliminary work on the development of adaptive strategies for parallel partitioning and linear solvers partially motivates this CQoS research.

Adaptive partitioning. We have employed a CQoS philosophy to improve the scalability of highly dynamic adaptive mesh refinement applications. As dynamic and localized features in the solution require higher mesh resolution for sufficient numerical accuracy, the mesh adapts dynamically to accommodate this. Parallel implementations of such adaptive applications present significant challenges in dynamic resource allocation as the overall efficiency is limited by the ability to partition the underlying mesh at runtime to expose all inherent parallelism, minimize communication and synchronization overheads, and balance load.

No partitioning technique performs the best for all combinations of application and computer system. Even worse, the best partitioner for a particular time frame of a parallel simulation might be the worst for the next time-frame [62]. Because the basic requirement for a partitioning method is dependent on the size and composition of the mesh, this requirement changes dynamically as the mesh adapts to the solution. By taking these dynamic conditions explicitly into account, the scalability for large, realistic simulations can be significantly improved. To meet the challenges in dynamic resource allocation inherent in parallel adaptive codes, we introduce another level of adaptation: *adaptive partitioning*, meaning dynamic and automatic switching of partitioning techniques, based on the current run-time state. The framework allowing this is called a *meta-partitioner*.

The meta-partitioner is a general framework for implementing adaptive partitioning in scientific, adaptive applications. It samples the current state of the application and the computer system in real-time and maps these samples onto a point in a partitioner-centric classification space (PCCS) [63,64]. This space is spanned by three axes corresponding to the three primary trade-offs inherent in the partitioning problem: communication vs. load balance, quality vs. speed, and data migration optimization. The application sample is based on mathematical properties of the mesh, while the sample of the computer system is based on current resource use and availability. The samples are translated into penalties that determine the location in the PCCS. Based on the location, one of the pre-characterized partitioning algorithms is selected and configured. This requires a set of partitioners to be thoroughly tested and characterized with respect to input parameters versus output quality metrics such as load imbalance and communication [37]. Our collaborators at Uppsala University in Sweden are characterizing the algorithms in the partitioning library `Nature+Fable` [62] by analyzing the complex-metric results from millions of partitioned real-world meshes.

Adaptive linear solvers. This research in CQoS has been partially motivated by large-scale scientific simulations based on partial differential equations [33,49], with emphasis on multimethod linear solvers in the context of parallel computational fluid dynamics, including flow in a driven cavity and compressible Euler flow (see [51] for details). Both applications employ pseudo-transient continuation with Newton methods, where the linearized Newton systems become progressively more difficult to solve as the simulation advances due to the use of pseudo-transient continuation [40]. Consequently both are good candidates for the use of adaptive linear solvers [9,10,50] where the goal is to improve overall performance by combining more robust (but more costly) methods when needed in a particularly challenging phase of solution with faster (though less powerful) methods in other phases. We designed parallel adaptive solvers with the goal of reducing the overall execution time of the simulation by dynamically selecting the most appropriate method to match the characteristics of the current linear system. A key facet of developing adaptive methods is the ability to consistently collect and access both runtime and historical performance data. We implemented a prototype component infrastructure that supports performance monitoring, analysis, and adaptation of important numerical kernels, such as nonlinear and linear system solvers [51]. We defined a simple, flexible interface for the implementation of adaptive nonlinear and linear solver heuristics. We also provide components for monitoring (based on TAU), checkpointing, and gathering of performance data.

6.2 Performance Measurement and Analysis

The TAU [58] performance system, particularly as applied to performance engineering technology for component software, provides a solid foundation for CQoS measurement and analysis infrastructure. Instrumentation and measurement of the performance of parallel applications and the analysis of multi-experiment performance data make it possible to characterize the performance for software variants of a parallel code over a range of runtime and problem parameters. From this information, models of performance for CQoS purposes can be created. However, the quality of the models strongly depends on the robustness of the performance system and its integration in a parallel programming methodology.

In addition to extensions of TAU's instrumentation and measurement capabilities, two advances in performance data analysis are important preliminary results. CQoS will depend on the value of performance knowledge obtained. The Performance Data Management Framework (PerfDMF) [34] was developed to provide for management and query of multi-experiment parallel profile data. PerfDMF enables the population of a multidimensional performance space from experiments and the analysis across data sets that lead to CQoS model creation. The Performance Explorer (PerfExplorer) tool [35] is a performance data mining framework that uses PerfDMF. PerfExplorer supports several analysis types, including comparative, cluster, dimension reduction, and correlation analysis. PerfDMF and PerfExplorer will provide the basis for developing CQoS infrastructure for performance analysis.

The second area of preliminary work reflecting support for CQoS research and development in CCA is our extension of TAU for performance monitoring of CCA component software [46]. TAU now provides the ability to instrument and measure the performance of CCA components and applications through a CCA-compatible performance component with its event creation and measurement interface, and component measurement ports and proxies [67]. A CCA performance monitor component and "mastermind" component were created to demonstrate runtime query of component performance state and modeling [55].

In addition, recent enhancements have been made in TAU's measurement system to allow more observation of functional aspects of a code's execution. To the extent that the measurement infrastructure can manage both functional and performance data, this infrastructure will be important for the integration of multiple CQoS metrics.

6.3 Software Quality through Contracts

Additional work has focused on software quality through runtime verification of basic interface contracts. Semantic annotations — integrated into the SIDL/Babel language interoperability toolkit [42] — are currently limited to the specification of constraints on the input and output of method calls and object properties. Efforts thus far have concentrated on exploring the impact on performance and failure detection effectiveness of a variety of traditional and experimental enforcement heuristics through simulation [19] and experimentation [19–21].

7 Conclusions and Future Work

This paper discussed some challenges in high-performance combustion, quantum chemistry, and accelerator simulations, each of which requires a means to compose, substitute, and reconfigure software so that tradeoffs can be made dynamically during runtime among performance, precision, underlying models, and reliability when choosing among available component implementations and parameters. We introduced our approach to tackling these issues by building infrastructure for computational quality of service (CQoS), including tools for both measurement/analysis and control infrastructure. We also discussed how component-based design and the Common Component Architecture provide a strong foundation for this work.

Our overall strategy for future work will be based on incremental progress in the design and implementation of CQoS support, initially with a focus on developing the CQoS testbed introduced in Section 5.4 with simplified representative cases drawn from the three motivating applications and exploring canonical problems. The middle and later stages of our work will extend to more complicated application scenarios; we will also increase the level of automation and make CQoS support more generally applicable to component applications.

Acknowledgments

This work has been supported in part by the U.S. Department of Energy's Scientific Discovery through Advanced Computing [68] initiative, through the Center for Component Technology for Terascale Simulation Software [3]. The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom we gratefully acknowledge. We especially thank Jim Amundson, Doug Dechow, Peter Stoltz, and Theresa Windus for discussing with us the CQoS needs of their simulations in high-energy accelerator physics and quantum chemistry.

References

1. B. Allan, R. Armstrong, S. Lefantzi, J. Ray, E. Walsh, and P. Wolfe. Ccaffeine – a CCA component framework for parallel computing. `http://www.cca-forum.org/ccafe/`, 2003.
2. Noriki Amano and Takuo Watanabe. A software model for flexible and safe adaptation of mobile code programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 57–61, Orlando, FL, May 2002.
3. R. Armstrong et al. Center for Component Technology for Terascale Simulation Software. `http://www.cca-forum.org/ccttss`, 2006.
4. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for high-performance scientific computing. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.
5. K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. Service-based software: The future for flexible software. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference (APSEC 2000)*, pages 214–221, 2000.
6. D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. Intl. J. High-Perf. Computing Appl., in press, 2006.
7. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.
8. S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of machine learning to selecting solvers for sparse linear systems. Presentation at the 2006 SIAM Conference on Parallel Processing, San Francisco, CA, February 2006.
9. S. Bhowmick, D. Kaushik, L. McInnes, B. Norris, and P. Raghavan. Parallel adaptive solvers in compressible PETSc-FUN3D simulations. Argonne National Laboratory preprint ANL/MCS-P1283-0805, 2005. submitted to Proc. of the 17th International Conference on Parallel CFD, Aug 2005.
10. S. Bhowmick, L. C. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in PDE-based simulations. In *Lecture Notes in Computer Science*, volume 2667, pages 828–839, 2003. Computational Science and its Applications-ICCSA 2003.
11. S. Bhowmick, P. Raghavan, L. C. McInnes, and B. Norris. *Faster PDE-Based Simulations Using Robust Composite Linear Solvers*, volume 20, pages 373–387. 2004.
12. G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, and C. C. Burt. A quality of service catalog for software components. In *Proceedings of the Southeastern Software Engineering Conference*. *`http://www.ndiatvc.org/SESEC2002/`*, 2002.
13. R. Bramley, D. Gannon, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Berg, S. Diwan, and M. Govindaraju. The Linear System Analyzer. In *Enabling Technologies for Computational Science*. Kluwer, 2000.
14. CCA Forum. CCA specification. `http://cca-forum.org/specification/`, 2006.
15. CCA Forum homepage. `http://www.cca-forum.org/`, 2006.
16. R. Chowdhary, P. Bhandarkar, and M. Parashar. Adaptive QoS management for collaboration in heterogeneous environments. In *Proceedings of the 16th International Parallel and Distributed Computing Symposium (IEEE, ACM), 11th Heterogeneous Computing Workshop*, Fort Lauderdale, FL, 2002.
17. D. Keyes (PI). Terascale Optimal PDE Simulations (TOPS) Center. `http://tops-scidac.org/`, 2006.
18. Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, CA, babel-0.9.4 edition, 2004.
19. Tamara L. Dahlgren. Adaptive enforcement of component interface assertions. Draft manuscript, 2006.
20. Tamara L. Dahlgren and Premkumar T. Devanbu. Adaptable assertion checking for scientific software components. In Philip M. Johnson, editor, *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS)*, pages 64–69, Edinburgh, Scotland, 24 May 2004. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-202898.

21. Tamara L. Dahlgren and Premkumar T. Devanbu. Improving scientific software component quality through assertions. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 73–77, St. Louis, Missouri, May 2005. Also available as Lawrence Livermore National Laboratory Technical Report UCRL-CONF-211000, Livermore, CA, 2005.
22. J. D. de St. Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A massively parallel prolem solving environment. In *Proceedings of the Ninth IEEE International Symposium on High Performance and Distributed Computing*, August 2000.
23. J. D. de St. Germain, A. Morris, S. G. Parker, A. D. Malony, and S. Shende. Integrating performance analysis in the Uintah software development cycle. In *Fourth International Symposium on High Performance Computing (ISHPC-IV)*, pages 190–206, May 15-17 2002.
24. Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science*, 2003.
25. Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *International Journal of High Performance Computing Applications*, 17:125–131, 2003. also Lapack Working Note 157, ICL-UT-02-07.
26. John W. Eaton. Octave. `http://www.octave.org`.
27. Thomas Eidson, Jack Dongarra, and Victor Eijkhout. Applying aspect-orient programming concepts to a component-based programming model. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS) April 22–26, 2003, Nice, France*, 2003.
28. V. Eijkhout and E. Fuentes. A proposed standard for matrix metadata. Technical Report ICL-UT 03-02, University of Tennessee, 2003.
29. M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the 9th International Workshop on Software Specification and Design*, pages 50–59, April 1998.
30. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of component-based applications within a Grid environment. In *Proceedings of SC2001*, 2001.
31. X. Gu and K. Nahrstedt. A scalable QoS-aware service aggregation model for peer-to-peer computing Grids. In *Proceedings of HPDC 2002*, 2002.
32. E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C. E. Houstis. A knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software*, 26(2):227–253, 2000.
33. P. Hovland, K. Keahey, L. C. McInnes, B. Norris, L. F. Diachin, and P. Raghavan. A quality of service approach for high-performance numerical components. In *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference*, Toulouse, France, 2003. Also available as Argonne National Laboratory preprint ANL/MCS-P1028-0203 via `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P1028.pdf`.
34. K. Huck, A. Malony, R. Bell, and A. Morris. Design and implementation of a parallel performance data management framework. In *Proceedings of the International Conference on Parallel Processing (ICPP 2005)*. IEEE Computer Society, 2005.
35. Kevin Huck and Allen Malony. PerfExplorer: A performance data mining framework for large scale parallel computing. In *Proceedings of SC—05*. ACM, November 2005.
36. R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
37. Henrik Johansson and Johan Steensland. A characterization of a hybrid and dynamic partitioner for SAMR applications. In *Proceedings of 16th IASTED International Conference Parallel and Distributed Computing and Systems*. ACTA Press, 2004.
38. K. Keahey, P. Beckman, and J. Ahrens. Ligature: A component architecture for high-performance applications. *International Journal of High-Performance Computing Applications*, (14), 2000.
39. Peter J. Keleher, Jeffrey K. Hollingsworth, and Dejan Perkovic. Exploiting application alternatives. In *19th International Conference on Distributed Computing Systems*, 1999.
40. C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM Journal on Numerical Analysis*, 35:508–523, 1998.
41. Joseph P. Kenny, Steven J. Benson, Yuri Alexeev, Jason Sarich, Curtis L. Janssen, Lois Curfman McInnes, Manojkumar Krishnan, Jarek Nieplocha, Elizabeth Jurrus, Carl Fahlstrom, and Theresa L. Windus. Component-based integration of chemistry and optimization software. *Journal of Computational Chemistry*, 24(14):1717–1725, 15 November 2004.
42. Lawrence Livermore National Laboratory. Babel. `http://www.llnl.gov/CASC/components/babel.html`.
43. Benjamin C. Lee, Richard Vuduc, James Demmel, and Katherine Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the International Conference on Parallel Processing*, Montreal, Quebec, Canada, August 2004.
44. H. Liu and M. Parashar. Enabling self-management of component based high-performance scientific applications. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, July 2005.
45. J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of ISORC '98*, 1998.

46. A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, C. Rasmussen, and M. Sottile. Performance technology for parallel and distributed component software. *Concurrency and Computation: Practice and Experience*, 17:117–141, Feb–Apr 2005.
47. Michael O. McCracken, Allan Snavely, and Allen Malony. Performance modeling for dynamic algorithm selection. In *Proc. of the International Conference on Computational Science (ICCS'03), LNCS*, volume 2660, pages 749–758, Berlin, 2003. Springer.
48. M. D. McIlroy. Mass produced software components. In *Proceedings of the NATO Software Engineering Conference*, pages 138–155, October 1968. Also available at `http://cm.bell-labs.com/cm/who/doug/components.txt`.
49. L. C. McInnes, B. A. Allan, R. Armstrong, S. J. Benson, D. E. Bernholdt, T. L. Dahlgren, L. F. Diachin, M. Krishnan, J. A. Kohl, J. W. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, and S. Zhou. Parallel PDE-based simulations using the Common Component Architecture. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 327–384. Springer, 2006. Also available as ANL/MCS-P1179-0704 via `http://www.mcs.anl.gov/cca/publications/p1179.pdf`.
50. L. C. McInnes, B. Norris, S. Bhowmick, and P. Raghavan. Adaptive sparse linear solvers for implicit CFD using Newton-Krylov algorithms. In *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics, June 17-20, 2003, Cambridge, MA*, volume 2, pages 1024–1028. Elsevier, 2003.
51. B. Norris, L. McInnes, and I. Veljkovic. Computational quality of service in parallel CFD. Argonne National Laboratory preprint ANL/MCS-P1283-0805, 2005. submitted to Proc. of the 17th International Conference on Parallel CFD, Aug 2005.
52. B. Norris, J. Ray, R. Armstrong, L. C. McInnes, D. E. Bernholdt, W. R. Elwasif, A. D. Malony, and S. Shende. Computational quality of service for scientific components. In *Proc. of International Symposium on Component-Based Software Engineering (CBSE7), Edinburgh, Scotland*, 2004. Also available as Argonne National Laboratory preprint ANL/MCS-P1131-0204 via `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P1131.pdf`.
53. H. Najm (PI). Computational facility for reacting flow science (CFRFS). `http://cfrfs.ca.sandia.gov`, 2006.
54. R. Raje, B. Bryant, A. Olson, M. Augoston, and C. Burt. A quality-of-service-based framework for creating distributed heterogeneous software components. *Concurrency Comput: Pract. Exper.*, (14):1009–1034, 2002.
55. J. Ray, N. Trebon, S. Shende, R. C. Armstrong, and A. Malony. Performance measurement and modeling of component applications in a high performance computing environment : A case study. In *Proceedings of the 18th International Parallel and Distributed Computing Symposium*, April 2003.
56. David Reiner and Tad Pinkerton. A method for adaptive performance improvement of operating systems. In *Proceedings of the 1981 ACM SIGMETRICS Conference on Measurement and Methodology of Computer Systems*, pages 2–10, September 1981.
57. Self-Adapting Large-scale Solver Architecture, see `http://icl.cs.utk.edu/salsa`, 2006.
58. S. Shende and A. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, Spring 2006.
59. Shweta Sinha and Manish Parashar. System sensitive runtime management of adaptive applications. In *Proceedings of the Tenth IEEE Heterogeneous Computing Workshop*, San Francisco, CA, 2001.
60. B. Smith et al. TOPS Solver Components. `http://www.mcs.anl.gov/scidac-tops/solver-components/tops.html`, 2005.
61. M. Sosonkina. Runtime adaptation of an iterative linear system solution to distributed environments. In *Applied Parallel Computing, PARA'2000*, volume 1947 of *Lecture Notes in Computer Science*, pages 132–140, Berlin, 2001. Springer-Verlag.
62. J. Steensland. *Efficient partitioning of dynamic structured grid hierarchies*. PhD thesis, University of Uppsala, Uppsala University Library, Box 510, SE-751, 20 Uppsala, Sweden, 2002.
63. Johan Steensland and Jaideep Ray. A partitioner-centric model for SAMR partitioning trade-off optimization: Part II. In *Proceedings of the 6th International Workshop on High Performance Scientific and Engineering Computing (HPSEC-04)*, August 2004. Held in conjunction with The 2004 International Conference On Parallel processing (ICPP-04), in Montreal, Canada.
64. Johan Steensland and Jaideep Ray. A partitioner-centric model for SAMR partitioning trade-off optimization: Part I. *International Journal of High Performance Computing Applications*, 19:1–14, 2005.
65. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, 1999.
66. Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards automated performance tuning. In *Proceedings of SC02*, 2002.
67. N. Trebon, A. Morris, J. Ray, S. Shende, and A. Malony. Performance modeling of component assemblies with TAU. Presented at Compframe 2005 workshop, Atlanta, June, 2005.
68. U. S. Dept. of Energy. SciDAC Initiative homepage. `http://www.osti.gov/scidac/`, 2006.
69. Jeffrey S. Vetter and Patrick H. Worley. Asserting performance expectations. In *Proceedings of SC02*, 2002.
70. Richard Vuduc, James Demmel, and Jeff Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, February 2004.
71. Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

72. R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.
73. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).
74. K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. A foundation for adaptive fault tolerance in software. In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 252–260, April 7-10, 2003.
75. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Second edition, 2005.
76. Eric Wohlstadter, Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, and Premkumar Devanbu. GlueQoS: Middleware to sweeten quality-of-service policy interactions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 189–199, May 23-28, 2004.
77. K. Zhang, K. Damevski, V. Venkatachalapathy, and S. Parker. SCIRun2: A CCA framework for high performance computing. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, Santa Fe, NM, April 2004. IEEE Press. to appear.