

Injecting Software Architectural Constraints into Legacy Scientific Applications

David Woollard^{1,2}

Chris Mattmann^{1,2}

Nenad Medvidovic²

¹Jet Propulsion Laboratory
4800 Oak Grove Drive, MS 171-264
Pasadena, California 91109, USA

Email: {woollard, mattmann}@jpl.nasa.gov

²Computer Science Department
University of Southern California
Los Angeles, California 90089, USA

Email: neno@usc.edu

Abstract

While software architectures have been shown to aid developers in maintenance, reuse, and evolution as well as many other software engineering tasks, there is little language-level support for these architectural concepts in scientific programming languages such as Fortran and C. Because many existing scientific codes are written in legacy languages, it is difficult to integrate them into architected software systems. By wrapping scientific codes in architecturally-aware interfaces, we are able to componentize legacy programs, integrating them into systems built with first-class architectural elements while meeting performance and throughput requirements of scientific codes.

1 Introduction

Scientific applications are complex, numerically intense codes which require significant new algorithm development and are often extremely long-lived due to this development difficulty. As a result, much of the scientific software in use today is written in Fortran and C, and these have remained the languages of choice for computational sciences.

While Fortran and C have many benefits to the scientific community including being pervasive, having highly-optimized compilers, and being generally applicable to numerically intensive algorithms, there are considerable drawbacks to this choice from a software engineering perspective. Encapsulation is a core building block onto which many modern software engineering principles are based, and is only moderately supported in Fortran and C [6]. Additionally, separation of concerns, as achieved via aspects [9] or first-class connectors [13], is fundamental to large-scale software development though difficult to build in these legacy languages.

One growing area of research emerging from the field of software engineering that has shown considerable worth in

not only the construction, but also the maintenance and evolution of complex software is *software architectures* [18]. Software architecture functions as the blueprint of a software system, abstracting the system into constituent elements such as components (the computational units), connectors (the communications facilities between these components), and configurations (the organization of the components and connectors including their mapping to physical hosts).

Though design-level architectural constraints are greatly beneficial to software developers, the gap between these concepts and implementation-level code requires the developer to maintain a complex mental mapping between architectural elements on one hand and language elements on the other. Reifying these architectural constructs as first-class elements of the implemented software system would directly aid the developer in a critical way: not only are design choices fundamentally codified, but there is a maintainable traceability between design and implementation which would eliminate the possibility of architectural drift and erosion [18].

It is exceedingly difficult to implement first-class architectural constructs in legacy languages such as Fortran and C due to limited support for high-level concepts such as objects with internal state and separation of concerns. While these languages do not natively support architectural concepts, we propose that wrappers which provide architecturally-defined component interfaces can be used to componentize scientific codes. In turn, this will allow us to integrate them with architecturally-aware components to form complex software systems while still meeting the performance requirements of the scientific application.

The use of architected scientific codes has many benefits, as architecture has been shown to aid the developer at all stages in the software development lifecycle. For example, software architecture aids the developer in isolating likely change into a component or set of components in the system. In turn, the system is more maintainable over time.

This benefit can be directly translated to scientific software, where scientific developers are often interested in refining a particular algorithm but are also interested in the overall stability of the software system.

A second benefit of architecting scientific codes can be found in the use of first-class connectors. By agglomerating communications between computational units into connectors, developers of these scientific codes allow for the possibility of replacing the connectors in the system with connectors that facilitate communications over networks, effectively distributing the legacy software system. Furthermore, software architects have developed a host of analysis tools that could be brought to bare on this newly distributed software system, optimizing the deployment of the system to maximize (or minimize) functional or non-functional properties such as bandwidth utilization or robustness.

In the rest of this paper, we will motivate the need for support of first-class architectural elements in legacy scientific code with an example of ongoing work at NASA's Jet Propulsion Laboratory (JPL). We will then discuss related work, introduce our wrapper technology and explore its computational impact on high performance scientific codes in both computation time and spacetime. We will illustrate the impact of software architecture on scientific codes by discussing an experiment with alternative deployment architectures using a recently developed statistical model comparison software system at JPL. Finally, we will conclude with a discussion of planned research in the area of software architectures for legacy scientific codes.

2 Motivation

At JPL, scientists process data from space-based instruments including Earth-monitoring satellites, exploratory probes, planetary (i.e., Mars) rovers, and so on as part of complex systems known collectively as ground data systems. These systems incorporate not only multiple legacy scientific codes but also a broad set of functionality including data cataloging and archiving as well as workflow management.

A motivating example of scientific software that would benefit from software architecture is one of JPL's current missions, the Orbiting Carbon Observatory (OCO). An earth-orbiting satellite launching in 2009, OCO will monitor CO₂ levels around the world and correlate onboard spectrometers with ground-based instruments before delivering data to scientists. As part of the ground data system for OCO, the Pre-flight Instrument Testing Process Pipeline (OpsPipeline) is a system consisting of a number of science codes processing data from instruments in pipe-and-filter fashion, while also communicating with a Catalog and Archive Service (CAS). This system is shown in Figure 1.

A recent change in the requirements of the OpsPipeline

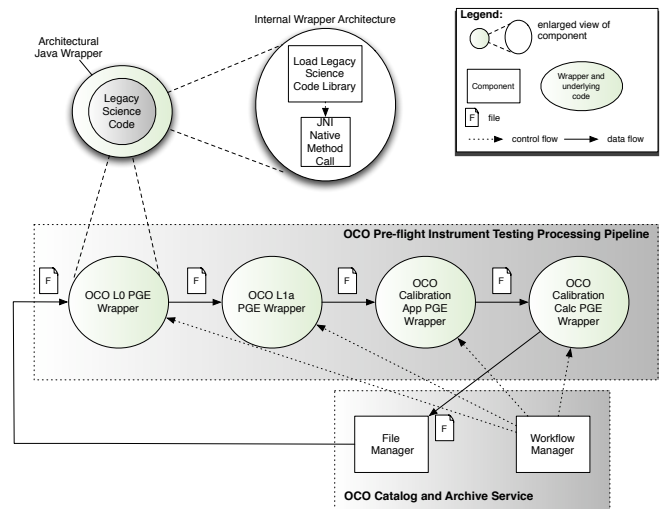


Figure 1. The OpsPipeline for the Orbiting Carbon Observatory's ground data system.

has forced us to look to processing portions of the pipeline that are computationally intensive (specifically, a physics-based retrieval algorithm for determining CO₂ concentrations based on spectral band intensity) remotely on third-party computational clusters. In this scenario, the change in the deployment architecture can be facilitated by the migration of the particular software component onto these remote hosts and replacement of this component's connectors with connectors that facilitate communications across the Internet through web-services mechanisms.

By fully integrating these scientific codes into our system architecture and enforcing architectural constraints, not only can we meet our goals for next generation ground data systems, but we can also provide system developers with means to evolve their software systems over time and address changing requirements through principles of architected software systems such as isolation of likely changes, separation of concerns, and increased modularity on the software system.

3 Related Work

There have been a number of initiatives to integrate software engineering principles with high performance scientific development, across a variety of domains, including high performance computing, grid computing (and more recently *cloud* computing). We will discuss representative projects in each of these areas, and along the way familiarize the reader with traditional jargon including software wrappers, component integration, and architectural middleware and adaptation.

Software wrappers [19] most often signify standard pieces of glue code that present a uniform external interface as a facade to an internal legacy application. Wrappers are frequently referred to as a special breed of *software connectors* [13], or a form of component integration, with recent emphasis being placed on automatically generating wrappers that fulfill performance requirements and behavioral guarantees (such as avoiding deadlock) [20]. Wrappers have also been studied in the information integration community as a means of automatically integrating information from semi-structured information sources [16]. In our work, we are currently focused on developing mappings between architectural wrappers and legacy scientific codes, at present using a manual process. Our future work includes the investigation of employing a graphical environment for automatically generating such wrapper interfaces using architectural knowledge and performance characteristics.

An example initiative leveraging the notion of component integration in the high performance computing domain is the Common Component Architecture (CCA) [2]. Developed by a consortium of universities and national laboratories, CCA seeks to define a common component interface and framework that will allow scientific developers to reuse code that has been written to the CCA standard. Though *architectural mismatch* [8] is of concern in CCA, the same is true of many standard component integration technologies (incl. CORBA [21], RMI [17], etc.) as component behavior (an influencing element in architectural mismatch) is traditionally not captured in an interface description employed in these technologies. In our work, we have chosen to leverage the PRISM-MW architectural middleware technology [11] (described below) as the architectural representation for our component wrappers; CCA and its scientific interface definition language (SIDL) may have been leveraged instead.

Other instances of component integration technologies for high performance computing lie in the realms of grid computing [7, 12], and more recently, cloud computing [10]. Grid computing is a software paradigm focused on the creation of *virtual organizations* sharing resources including storage, processing, and user identities in the support of large-scale scientific computation. Grids are primarily computation-focused, though there are a few successful storage-focused grid systems. Cloud computing, on the other hand, is concerned with the utilization of lowest-common-denominator services, compute and store, and their availability and provision as a service commodity including a standard external interface. In cloud computing, users pay for levels of service for functional properties like scalability, efficiency, and reliability, and can then pay to increase those levels on demand. Most grid environments require the ability to execute legacy scientific code, making

our wrapper integration technology a prime candidate for infusing architectural understanding into grid systems. In addition, cloud computing can serve as a suitable platform for delivering scalability and efficiency to legacy scientific codes, as integrating cloud services with our legacy scientific wrappers can be performed through standard API calls at the wrapper level.

In the software architectural domain, architects have developed a handful of techniques for integrating the high-level concepts of software architectures with underlying implementations. Architectural description languages (ADLs) [14], for example, have been used to define, describe, and validate software architectures. These languages have focused primarily on model checking in terms of interface correctness and other system-level properties such as deadlock freedom. While ADLs occasionally provide a mechanism for code generation, they do not support integration with legacy code or traceability to implementation artifacts. Two systems have tried to remedy this gap between architectural elements and implementation artifacts. ArchJava [1] is a programming language that provides the features of an ADL as a superset of Java. ArchJava thus requires a custom compiler, and limits the engineer to the natively supported set of architectural primitives.

Prism-MW[11] takes a different approach: it is an extensible, architecturally-aware middleware platform implemented in standard Java. Prism-MW uses a set of core classes to represent the basic architectural elements: *Architecture*, *Component*, *Connector*, and *Port*. It also provides an explicit extension mechanism (via Java's abstract classes and interfaces) to implement specific variants of these elements for use in different application scenarios and architectural styles [18, 15]. We use Prism-MW, and rely particularly on its extensibility mechanisms and native support for architectural styles, in our approach for implementing our legacy wrappers as discussed in the ensuing section.

4 Java Component Interface Wrappers

In this section, we will discuss the injection of architectural constraints in the form of Prism-MW wrappers applied to legacy Fortran scientific codes. We have accomplished this injection by instantiating legacy Fortran and C code through the Java Native Interface (JNI). Minor alterations of the legacy code are required in order to make it callable via JNI methods, though our experience has shown these modifications to be unobtrusive to the scientific algorithms, only requiring changes to the initialization and completion of the code.

Prism-MW, as described initially in the related works section, provides the developer with a means of programming software for distributed systems that enforces the software architectural constraints. Components and connectors

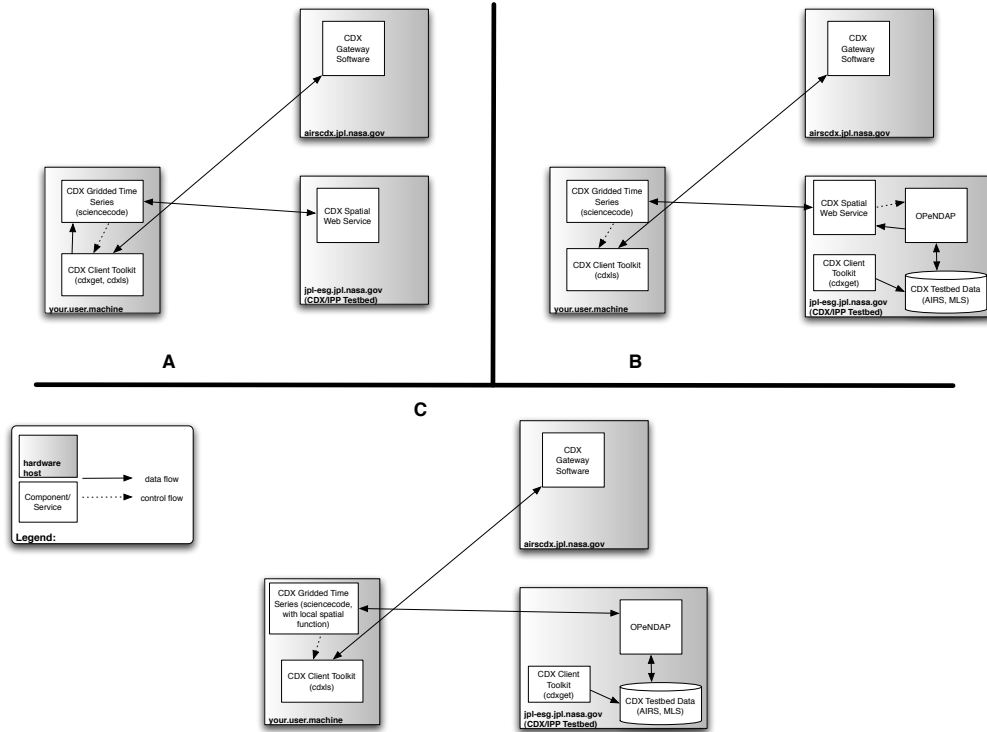


Figure 2. Architectural adaptability via our java wrappers.

are explicitly declared as part of an event-based software system that can be distributed among a multitude of hardware hosts. The class diagram for Prism-MW is presented in Figure 3.

Prism-MW uses an *Architecture* class to record the configuration of its constituent *Components*, *Connectors*, and *Ports*, and facilitate their addition, removal and reconfiguration. Distributed systems are implemented as sets of interactive architecture objects. Interaction between components in each architecture is facilitated by passing *Events* between *Components' Ports*. Each of these base classes has an *Extensible* counterpart to allow customization of Prism-MW to different application scenarios and architectural styles.

Prism-MW supports multiple styles by providing extensibility along five dimensions: structure, topology, behavior, interaction, and data. **Topological** and **structural** constraints are satisfied via implementations of the *AbstractTopology* class which is used by the *ExtensibleArchitecture* class. Topological constraints, such as the pipeline architecture of OpsPipeline in Figure 1, can be enforced via a child class of *AbstractTopology*.

To specify style **interactions**, *ExtensibleConnectors* use *AbstractHandlers* to implement event routing policies. For example, the unidirectional data forwarding used by connectors in the OpsPipeline (see Figure 1) can be developed as implementations of the *AbstractHandler* class.

Style-specific **data** constraints are implemented via *ExtensibleEvents*.

The final dimension of architectural variance, **behavior**, is implemented via *ExtensibleComponent*. In order to componentize legacy scientific code, each Java wrapper is an extension of this *ExtensibleComponent* class, containing *Ports* in order to facilitate communications and registering style information with the system's *ExtensibleArchitecture*.

The resulting system enforces the pipe-and-filter architectural style, including the interactions between PGEs and the structure of the OpsPipeline through implementations of Prism-MW's abstract classes. The science algorithms implemented in legacy code remain untouched while the wrapper handles integration into the ground data system, including registration with the *ExtensibleArchitecture*, event handling and buffering, and other architectural constraints needed for the OpsPipeline to conform to a modified pipe-and-filter style.

5 Architectural Benefits

As part of the Climate Data eXchange (CDX) project at JPL [5], we have been involved in developing software to integrate legacy atmospheric observational data sets from several NASA data holdings at JPL, and to make that data

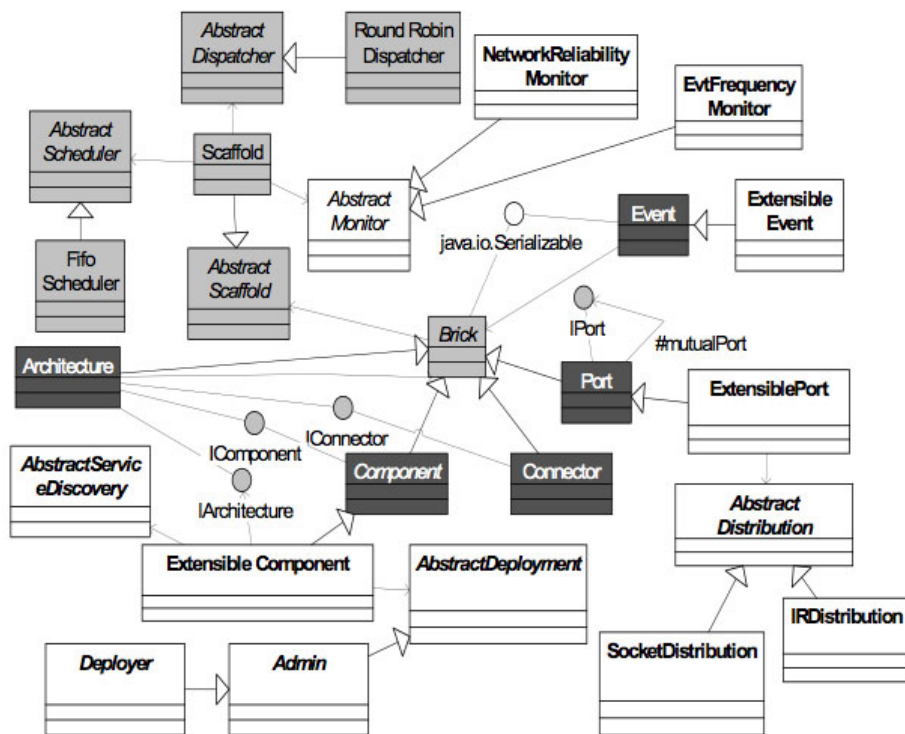


Figure 3. UML class diagram of Prism-MW showing its core in grey and extensible classes in white.

available for comparison to climate model output, in order to statistically grade both the observational data and the models on which critical earth processes such as weather understanding and climate change understanding are based.

To evaluate the architectural benefits of our java wrappers, we constructed a time series grid statistical simulation spatially and temporally comparing and scoring climate model output from the DOE-funded Earth System Grid project [3]. The ESG runs large-scale climate models and produces model output data that can be compared with observational data sets taken from the same spatial region over the same temporal constraints to perform model scoring and intercomparison. For the purposes of our prototype, data sets from NASA's Atmospheric Infrared Sounder (AIRS) mission were leveraged. AIRS data includes climate related measured variables such as water vapor, and standard air pressure.

Fig. 2 demonstrates a comparison of three architectures (labeled A, B and C in the diagram) leveraged to drive the implementation of our simulation software. In all three variations, the *CDX Gridded Time Series* was the driver program responsible for orchestrating the AIRS and ESG data assimilation (from the *CDX Gateway component*), and then evaluating the acquired datasets for spatial and temporal resolution using a legacy function, wrapped using our

java wrappers (the *CDX Spatial Web Service component* in Fig. 2).

Fig. 2-A demonstrates the first architectural variation in which all remote AIRS climate datasets within a time range are pulled down to the user machine using CDX client software and spatially evaluated using a remotely accessible Java wrapper. Fig. 2-B demonstrates the same functional capability, with the exception of extending the spatial Java wrapper for remote dataset evaluation, using a data subsetting server technology called OPeNDAP [4]. Fig. 2-C demonstrates the final architecture, integrating the wrapped spatial evaluation functionality into the gridded time series driver program on the client machine.

The three architectural variations, while seemingly the same set of components, and connections, shifted around, exhibit decidedly different functional properties and implementations. In the first option (A), all the data is pulled down to the local client machine (using the *CDX Client Tools*, *cdxget* and *cdxls*, which pull data, and remotely list it, respectively). This is by far the slowest, and most cumbersome implementation of our time series calculation. The second option (B), provides an architecture that reduces the amount of data that needs to be pulled down by allowing remote spatial evaluation to access AIRS datasets remotely, without pulling them down to the client machine using *CDX*

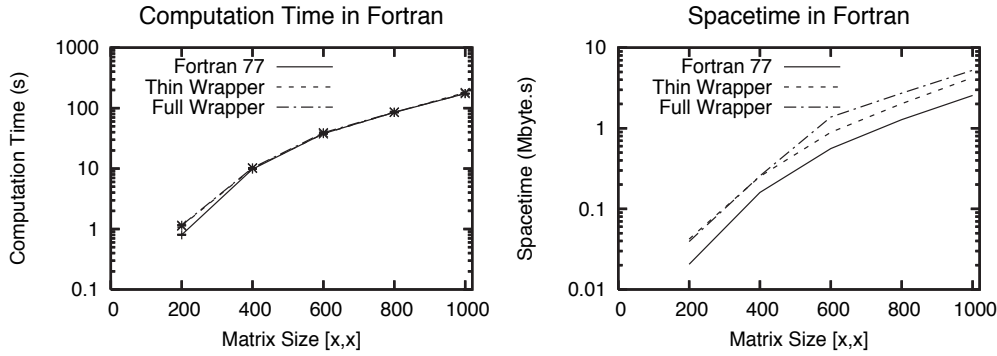


Figure 4. The performance impact of architectural constraint wrappers on Fortran codes.

Client Tools. The final option (C), provides an architecture in which spatial evaluation occurs on the client machine, and data access occurs remotely (and only subsets of data are sent over the network, as required by the wrapped spatial evaluation function).

Architectural option C reduces the amount of time to perform time series grid calculation and model comparison/scoring from 9 hours (in the case of option A) to a few minutes, improving efficiency and effectiveness of computing the time series grid. The use of our architectural wrappers in this scenario enabled our CDX software to re-locate (in each architectural option), the CDX spatial computation service from a remote service (in which a subset of observation data is sent from the client machine over the network to the spatial service) all the way to moving the spatial service to the client machine and then requesting subsets of remote data and limiting the bandwidth and time required to compute our output time series, improving upon our initial architecture and implementation by a factor of 500.

6 Performance and Wrapper Overhead

Scientific application developers are concerned not only with the accuracy of their code, but also with the code’s performance. Large data sets and high precision require intense computation. In injecting architectural constraints into these legacy scientific codes, we have incurred a performance overhead which we must consider.

In order to measure the performance overhead of own Prism-MW wrappers on Fortran and C code, we used a canonical scientific software metric—the multiplication of two matrices. Starting with legacy base applications in both C and Fortran 77, we developed two types of Java wrappers: a “Thin Wrapper” which simply executes the legacy code without passing it any parameters or getting any in re-

turn, and a “Full Wrapper” which passes the legacy code matrices to be multiplied and gets the resulting matrix of the calculation when the legacy code returns.

The results obtained from the Fortran full and thin wrappers in both execution time and spacetime are given in Figure 4. These results closely match our performance results for C wrappers. While the execution time overhead incurred by both the thin and the full wrapper is significant for shorter executions (approximately 40% execution time overhead for multiplying two matrices of 200×200 doubles), this overhead is insignificant for larger computations (the overhead for the multiplication of 1000×1000 doubles is under 2%). This decrease in overhead is a result of the amortization of the relatively constant cost of the Java Virtual Machine (JVM). In addition, we found that execution time is unaffected by the passing of data from the java wrapper to legacy code (i.e., the execution time difference between our thin wrapper and full wrapper is negligible).

In terms of memory performance, both java wrappers incur overhead due not only to the instantiation of the JVM, but also due to memory allocations in the Java portion of the wrapper. Since data must be passed through native methods by value, additional overhead is incurred by the full wrapper: the full wrapper occupies approximately double the memory of the original code due to its need to allocate matching space for parameters.

This experiment shows that granularity of the computation and the ratio of computation to data movement should always be considered, especially in order to amortize the costs of increased memory usage. Existing literature suggests that a performance penalty of more than 10% execution time overhead in the high-performance scientific community is too great a price to pay for increased functionality [2]. In our example of dense matrix multiplication, we have shown that architectural wrappers are well within this upper

bound. Though our spacetime performance is more obtrusive, we feel that for many applications, including our work at JPL, this penalty is acceptable.

7 Conclusion

Injecting software architectural constraints into legacy scientific codes allows software developers at JPL to fully incorporate legacy applications into modern, architected software systems. As shown in this paper, an immediate benefit of architecting these systems is the ability to adapt the resulting software system to multiple deployment architectures, improving efficiency and performance. Over time, our experience has shown that architecture impacts not only design decisions but other portions of the software lifecycle as well, including maintenance, evolution, and reuse.

While performance of our architected software might have been a concern to potential practitioners, we have shown in this paper that performance should not be a primary concern. Our proposed scientific software component interfaces meet the needs of software engineers, reifying software architectural elements as artifacts in code, they also meet the strict performance requirements of scientific code developers, incurring only minimal computational overhead.

In our future work, we propose to utilize deployment architecture analysis tools to further optimize the deployment of software systems such as the OCO's OpsPipeline and our CDX model scoring system into heterogeneous hardware environments to increase system throughput and robustness.

Acknowledgment

This work has been partially sponsored by the National Science Foundation under Grant number ITR-0312780 and by the Jet Propulsion Laboratory, managed by the California Institute of Technology, under contract with NASA.

References

- [1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *ICSE*, 2002.
- [2] B. A. Allen et al. A component architecture for high-performance scientific computing. *The International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.
- [3] D. E. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. L. Chervenak, L. Cinquini, B. Drach, I. T. Foster, P. Fox, J. Garcia, C. Kesselman, R. S. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams. The earth system grid: Supporting the next generation of climate modeling research. *CoRR*, abs/0712.2262, 2007. informal publication.
- [4] P. Cornillon, J. Gallagher, and T. Sgouros. Opendap: Accessing data in a distributed, heterogeneous environment. *Data Science Journal*, 2:164–174, 2003.
- [5] D. Crichton, C. Mattmann, and A. Braverman. Facilitating climate modeling research and analysis via the climate data exchange. In *Workshop on Global Organization for Earth System Science Portals (GO-ESSP)*, 2008.
- [6] V. K. Decyk, C. D. Norton, and B. K. Szymanski. How to express c++ concepts in fortran 90. *Scientific Programming*, 6(4):363, 1997.
- [7] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *J. Supercomputing Applications*, pages 1–25, 2001.
- [8] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
- [9] G. Kiczales et al. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
- [10] N. Leavitt. Is cloud computing really ready for prime time? *Computer*, 42(1):15–20, 2009.
- [11] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Trans. Software Eng.*, 31(3):256–272, 2005.
- [12] C. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In *ICSE*, pages 721–730, 2006.
- [13] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE*, pages 178–187, 2000.
- [14] N. Mevidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.
- [15] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *First Workshop on Self-Healing Systems*, 2002.
- [16] I. Muslea, S. Minton, and C. A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93–114, 2001.
- [17] C. Nester, M. Philippsen, and B. Haumacher. A more efficient rmi for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, New York, NY, USA, 1999. ACM.
- [18] D. E. Perry and A. L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, October, 1992.
- [19] H. M. Sneed. The rationale for software wrapping. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 303, Washington, DC, USA, 1997. IEEE Computer Society.
- [20] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *ICSE*, pages 374–384, 2003.
- [21] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14:46–55, 1997.