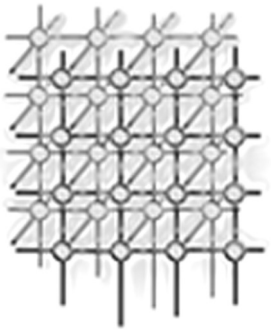
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1076



Performance modeling of component assemblies

Nick Trebon[1,2,*,†], Allen Morris[1], Jaideep Ray[3], Sameer Shende[1] and A. D. Malony[1]

[1]*Department of Computer Science, University of Oregon, Eugene, OR 97403-1202, U.S.A.*
[2]*Department of Computer Science, University of Chicago, Chicago, IL 60637, U.S.A.*
[3]*Sandia National Laboratories, Livermore, CA 94551-0969, U.S.A.*

SUMMARY

A parallel component environment places constraints on performance measurement and modeling. For instance, it must be possible to instrument the application without access to the source code. In addition, a component may admit multiple implementations, based on the choice of algorithm, data structure, parallelization strategy, etc., posing the user with the problem of having to choose the 'correct' implementation and achieve an optimal (fastest) component assembly. Under the assumption that an empirical performance model exists for each implementation of each component, simply choosing the optimal implementation of each component does not guarantee an optimal component assembly since components interact with each other. An optimal solution may be obtained by evaluating the performance of all of the possible realizations of a component assembly given the components and all of their implementations, but the exponential complexity renders the approach unfeasible as the number of components and their implementations rise. This paper describes a non-intrusive, coarse-grained performance monitoring system that allows the user to gather performance data through the use of proxies. In addition, a simple optimization library that identifies a nearly optimal configuration is proposed. Finally, some experimental results are presented that illustrate the measurement and optimization strategies. Copyright © 2006 John Wiley & Sons, Ltd.

Received 29 August 2005; Revised 8 February 2006; Accepted 9 March 2006

KEY WORDS: component; performance; TAU

*Correspondence to: Nick Trebon, Department of Computer Science, 120 Deschutes Hall, 1020 University of Oregon, Eugene, OR 97403-1202, U.S.A.
†E-mail: ntrebon@cs.uoregon.edu

Contract/grant sponsor: U.S. Department of Energy, Office of Science; contract/grant numbers: DE-FG03-01ER25501 and DE-FG02-05ER25680
Contract/grant sponsor: United States Department of Energy's National Nuclear Security Administration; contract/grant number: DE-AC04-94AL85000

1. INTRODUCTION

The Common Component Architecture (CCA) is a component-based methodology for developing scientific simulation codes. This architecture consists of a *framework* that enables *components* (embodiments of numerical algorithms and physical models) to work together. Components publish their interfaces and use interfaces published by others. Access to functionality is allowed through ports. Once components are instantiated and registered with the framework, the process of connecting ports is just the movement of (pointers to) interfaces from the *providing* component to the *using* component. A method invocation on a UsesPort thus incurs a virtual function call overhead before the actual implemented method is used. Components publishing the same interface and with the same functionality (but perhaps implemented via a different algorithm or data structure) may be transparently substituted for each other in a *code* or a *component assembly*. Components are compiled into shared libraries and are loaded in, instantiated and composed into a useful code at runtime. Details regarding the CCA can be found in [1,2]. An analysis of the process of decomposing a legacy simulation code and re-synthesizing it as components can be found in [3,4]. Actual scientific results obtained from this toolkit can be found in [5,6].

One of the benefits of components is their reusability. As a result, the component writer is rarely the sole user of the components. The correctness of a component is the responsibility of the component writer. The performance, however, is of primary importance to the component user who may not be familiar with the implementation of the component. Thus, in the context of high performance computing (HPC) with components, one needs a reliable way of measuring the performance of each implementation of a component. However, performance measurement in a component environment introduces several challenges not found in traditional monolithic codes. First, the overall performance of the component application is comprised of the performance of the individual components and the performance of the interaction between components. In addition, a single component may be realized through multiple implementations (i.e. components that provide the same functionality but implement a different algorithm or utilize a different internal data structure). An arbitrary selection of one implementation may result in a correct but suboptimal global solution. Finally, direct instrumentation of the source code may not be possible, as the component user may not have access to the source code. These requirements necessitate a non-intrusive, coarse-grained measurement framework that can instrument an application at the component level.

Once an empirical performance model has been created for each of the components and their implementations, it is possible to construct a performance model for the composite application. By comparing global performance models utilizing different subsets of component implementations, it is possible to select an optimal set of implementations for a given problem. However, simply selecting the optimal implementation of each component does not guarantee an optimal global solution because the individual performance models do not consider the interactions between components (e.g. cost of translating one data structure to another if the interacting components have different data layouts). If the number of components and their implementations are small, one may adopt the brute-force approach of enumerating all the possible realizations of the component assembly, constructing and evaluating their composite performance models, and choosing the optimal one. However, typical scientific codes consist of assemblies of 10 to 20 components. If each component has three implementations, the solution space consists of 3^{10} to 3^{20} possible realizations, which makes the brute-force approach of comparing each realization unfeasible. This paper proposes a simple, heuristic-based algorithm that

DOI: 10.1002/cpe

identifies a nearly optimal configuration by reducing the overall solution space by identifying and eliminating 'insignificant' components from the optimization phase. A component is insignificant if it does not contribute significantly to the overall performance of the application (e.g. if the measured contribution of a component to the composite application is less than some threshold). Once an optimal configuration for the remaining 'core' components has been identified, a complete (and nearly optimal) configuration can be achieved by using (adding) any implementation of the insignificant components. This ensures that the final assembly will be close to optimal.

2. LITERATURE SURVEY

The CCA was designed specifically with the needs of high-performance computing in mind. The three most popular commodity component models, CORBA [7], Microsoft's COM/DCOM [8] and Enterprise Java Beans [9] are unsuitable because they lack support for efficient parallel communication, do not incorporate sufficient data abstractions (e.g. complex numbers or multi-dimensional arrays) and/or do not provide language interoperability. These commodity models are also designed for serial applications and, as a result, are not concerned with aspects of the underlying system that are critical to high-performance computing, such as the hardware details and memory hierarchy.

In spite of the shortcomings inherent in the commodity component models, similar approaches in *measuring* performance are valid. Proxy-based measurement infrastructures have been described for both Java Beans [10] and CORBA [11,12]. The Java infrastructure utilizes proxies and a monitor component to identify hotspots or components that do not scale well. The CORBA framework (WABASH) also consists of proxies and manager components. Here, the manager is responsible for querying the proxies for data retrieval and event management.

Runtime automated optimization of codes, as done by Autopilot [13,14] and Active Harmony [15] are closest to our approach as outlined in this paper. Both require the application to identify performance parameters (and the valid values that they can take) to a tuning infrastructure. This infrastructure also monitors their performance, which in the case of Active Harmony is provided by a function (the *objective* function) implemented by the application itself. Each of the parameters are perturbed and the application is executed to obtain the effect of the perturbation. Active Harmony relies upon a simplex algorithm to identify the optimal values of the parameters while Autopilot uses fuzzy logic. Both Active Harmony and Autopilot require the identification of 'bad' regions in parameter space, so that the optimization search may be concluded faster by avoiding these regions. Active Harmony also has an infrastructure to swap in/out multiple libraries in order to identify an optimal implementation.

In the realm of optimization of CCA applications, an extension is proposed that allows for the introduction of self-managing components [16]. This extension includes modifying individual components to include monitoring and control capabilities, as well as updating the underlying framework to support rule-based management actions. Two distinct manager component types are also defined: a component manager that determines intracomponent decisions, and a composition manager that administers intercomponent decisions. Component interaction is expressed and enforced through high-level rules.

Current research at Argonne National Laboratory is also very similar in essence [17]. Their research is focused on a software infrastructure for performance monitoring, performance data management and

DOI: 10.1002/cpe

adaptive algorithm development of parallel component PDE-based simulations. Specifically, their work seeks to optimize the linear and nonlinear solver components, which are responsible for the majority of the computation in PDE-based simulations. Similar to the work described in this paper, the TAU [18,19] library is utilized for instrumentation and performance data is stored in the PerfDMF database [20]. This research also attempts to incorporate a quality-of-service aspect. Application characteristics that affect performance, such as convergence rate, parallel scalability and accuracy are stored as metadata in a persistent database, and can be evaluated along with the performance model.

3. PERFORMANCE MEASUREMENT

The composite performance of a component application depends upon the performance of the individual components and the efficiency of their interactions. The performance of a given component is meaningless without knowing the *context* in which it is used. This context includes the problem being solved, the input(s) to the component, and the intercomponent interactions. Consider a component that performs two multidimensional array functions. The first function accesses the array in a sequential manner, the second function utilizes a strided access method. Clearly, the performance of the component depends upon the problem being solved (i.e. which function is being applied to the array). If the input to the component is modified (e.g. changing the size of the input array), the performance will also change. The final contextual requirement relates to the performance caused by the translation of mismatched data types.

From an optimization perspective, the component developer will be interested in optimizing the composite application, rather than individual components. As a result, a coarse-grained performance measurement infrastructure is needed. Based upon related work, a proxy-based measurement infrastructure for the CCA was proposed and implemented in [21]. A brief summary of this framework is presented below.

The framework consists of three component types. First, lightweight proxies are interposed between the caller and the callee components and trap any method calls, enabling performance measurements to be taken. The second component type is responsible for making the performance measurements, which includes timer creation and hardware interaction. The measurement system makes use of the tuning and analysis utilities (TAU) measurement library [18,19]. The final component type is the Mastermind, which is responsible for gathering, storing and reporting the measurement data. Each proxy is connected to the Mastermind component, which provides functionality to turn measurements on and off. The Mastermind, in turn, is connected to the TAU component, which provides the functionality to create and manage timers, as well as access hardware counters.

A simple example depicting the relationship between the framework component types is depicted in Figure 1. In this example there is a single component, C, being instrumented. C provides two ports, a and b. In order to instrument calls on these ports, the proxy uses and provides ports of the same type. The proxy also uses functionality provided by the Mastermind component, which enables the proxy to turn instrumentation on and off. In turn, the Mastermind uses the measurement functionality provided by TAU.

Because of the proxies' lightweight nature, their creation is a mechanical process. A supplemental tool has been developed that automatically generates a proxy for a given component. This tool is based on the Program Database Toolkit [22], which allows for analysis and processing of C++ source code. To simplify the automatic process, proxies are created on a per port basis rather than on a per

DOI: 10.1002/cpe

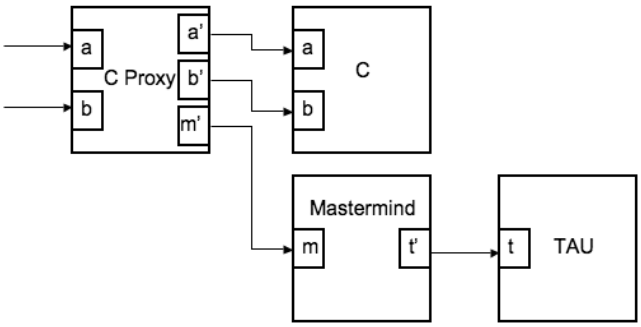


Figure 1. An example of the measurement framework. A proxy for component C instruments two ports (a and b) provided by C.

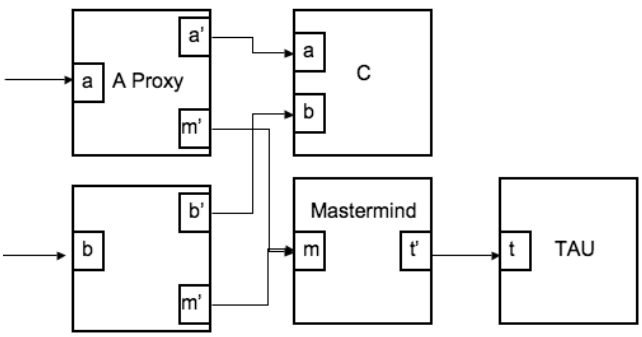


Figure 2. The measurement framework using one proxy per port.

component basis. In Figure 2, an example of this topology is pictured. The proxy generator also allows for proxies to be built in the absence of a Mastermind component. These proxies would simply insert timers directly from the TAU component. All timing data would then be dumped to disk in TAU profile files upon program termination.

In order to ease the storage and analysis of the measured data, the Performance Data Management Framework [20] can be utilized. This framework is developed along with TAU, and includes a database and visualization facilities.

4. AN APPROXIMATE APPROACH

Using the measurement infrastructure described in the previous sections, empirical performance models can be generated for a given component instance using a variety of methods, such as regression analysis. One can synthesize a global application by composing individual component models together

DOI: 10.1002/cpe

(as long as the context is preserved, i.e. component interactions are accounted for). This ability to generate an empirical composite performance model introduces the question of which set of component instances offers the best performance for a given problem. This section introduces a preliminary algorithm for identifying an approximately optimal solution. The optimization library is independent from the measurement framework, and is intended as the third phase of the measurement-modeling-optimization procedure.

A typical scientific application may contain as many as 10 to 20 components. If each component has three instances, then there are a total of 3^{10} to 3^{20} total configurations. Such a solution space makes the brute-force solution of enumerating and evaluating all possible realizations undesirable. Fortunately, typical scientific applications often contain solver components that are responsible for the majority of an application's computation. By identifying these 'core' components and eliminating the 'insignificant' components, a reduced solution space can be achieved. Once the core components are optimized, a complete configuration can be realized by arbitrarily including any implementation of the insignificant components. Because these components offer very little to the overall computation of the global application, a poor choice would have very little effect on the overall performance. Thus, an approximately optimal solution can be achieved.

In order to implement this approximate approach, a pruning application applies the pruning algorithm on the component call-graph of the application. It is important to note that a component may exist in multiple locations in the call-graph, as it may lie on multiple call-paths. Thus, a single instance may require multiple performance models as the context may be different between distinct call path locations (e.g. one call-path may require a data transformation, and another may not). The call-graph can be traversed, and based upon a set of rules, insignificant branches may be pruned. 'Insignificance' is determined by the inclusive time of a component, which is the cumulative time spent during the execution of any of the component's methods, including subroutine calls. The resulting graph will be an optimized core tree. Insignificance is determined by the algorithm presented below.

Let C_J represent the set of children of node J. Let T_i, $i \in C_J$ be the inclusive time of child i of node J. Let J have N_J children, i.e. the order of the finite set C_J is N_J. When examining the children of a given node, we see that two cases arise.

1. The total inclusive time of the children is insignificant compared to the inclusive time of node J, i.e. $\sum T_i / T_J (\forall i \in C_J) < \alpha$, where $0 < \alpha \ll 1$. Thus, the children contribute little to the parent node's performance and may be safely eliminated from further analysis. α is typically around 0.1, i.e. 10%.
2. The total inclusive time of the children is a substantial fraction of node J's inclusive time, i.e. the children contribute significantly $\sum T_i / T_J (\forall i \in C_J) \geq \alpha$. In this case the children are analyzed to identify if dominant siblings exist.

Let $T' = \sum T_i / N_J (\forall i \in C_J)$ be the average or 'representative' inclusive time for the elements of C_J. We then iterate through each node $i \in C_J$, eliminating the elements of $i \in C_J$ where $C_J / T' < \beta$. Thus, children of J whose contributions are small relative to a representative figure are eliminated. Typically, β is chosen to be around 0.1, i.e. 10%.

This algorithm describes two cases in which branches may be pruned. All pruning decisions are based on inclusive performance to avoid pruning an insignificant root that has significant children. First, if the total contribution of the children is less than some threshold, all the children may be safely pruned (step 1 of the pruning algorithm). The second heuristic is then applied to each child

DOI: 10.1002/cpe

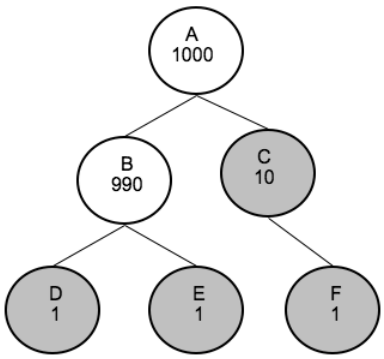


Figure 3. A simple call-graph example with the pruned nodes shaded. The internal values represent the inclusive performance contribution.

separately to determine if its contribution is small relative to the average contribution of its siblings. Pruning decisions are made relative to the contribution of the parent or siblings' rather than to the overall application performance in order to prevent pruning off significant children who contribute evenly. For example, consider an application that consists of a root with 15 children, each of whom contribute $1/15$ of the overall performance (the root contributes a negligible amount on top of the children). If the pruner makes decisions relative to the overall performance using a threshold of 0.1, all the children would be falsely identified as insignificant. At this stage, the thresholds are chosen by the user. For the purposes of our experiments, threshold values of 10% worked reasonably well.

Consider the pruning algorithm applied to the example call-graph depicted in Figure 3. The graph consists of six unique nodes. The inclusive performance is included in each node. For this example, threshold values of 0.1 are used for α and β. The pruning algorithm works in a depth-first manner, beginning at node A. Children B and C do contribute significantly, relative to A's inclusive contribution, so both are preserved. Each child is then examined in turn. The average contribution for B and C is 500. Thus, with a 0.1 threshold, node C is pruned (along with its children), as its' inclusive contribution is less than 50 (10% of 500). B is preserved, and its children are examined cumulatively. It is determined that their contribution is insignificant, and B's children are pruned. The pruned branches are shaded, leaving the resulting core tree behind.

5. EXPERIMENTAL RESULTS

In order to fully demonstrate the measurement and optimization phases described in the previous sections, several small experiments were conducted. By using 'dummy' components that implement a known performance model, both the measurement infrastructure and optimization algorithm can be tested and verified. Instead of performing actual 'work', these dummy components simply implement a known performance model based upon their input parameter(s). Thus, it is nearly trivial to create dummy components that perform to any specification.

DOI: 10.1002/cpe

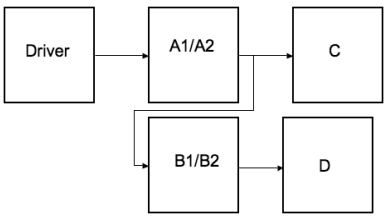


Figure 4. A simple component application diagram. Note that A and B both have multiple implementations (i.e. A1, A2, B1 and B2).

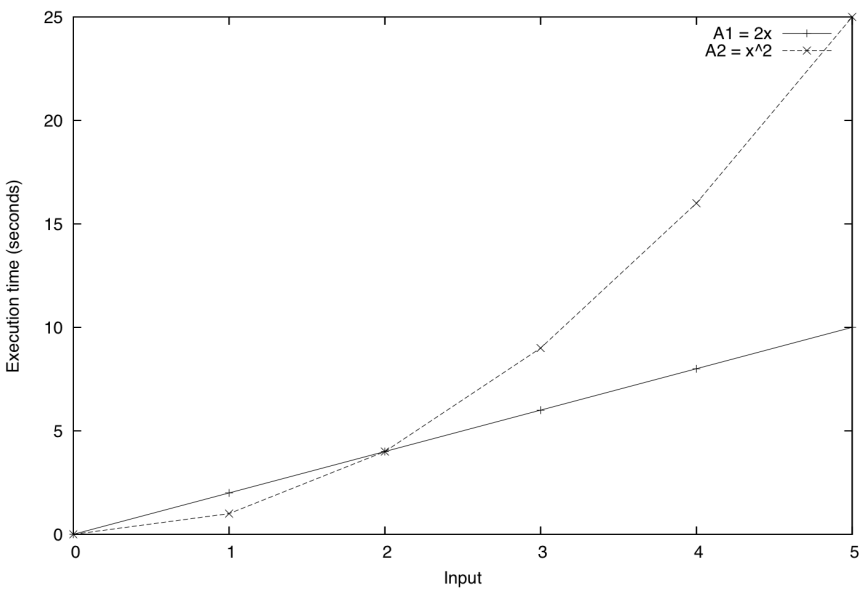


Figure 5. The performance models for both implementations of component A are plotted.

One example application used is depicted in Figure 4, which shows the application without any of the measurement infrastructure. Components A and B both have multiple implementations available that offer different performance. Components C and D are included as examples of 'insignificant' components.

Component A's implementations execute the performance models $y = 2x$ and $y = x^2$. These equations are plotted in Figure 5. Component B's implementations are represented by the equations $y = x^3$ and $y = 2x^2$. B's performance models are plotted in Figure 6. The models for components A and B were chosen because they have similar characteristics. Both have clear choices for when a given implementation should be chosen, but this choice depends upon the input. At input

DOI: 10.1002/cpe

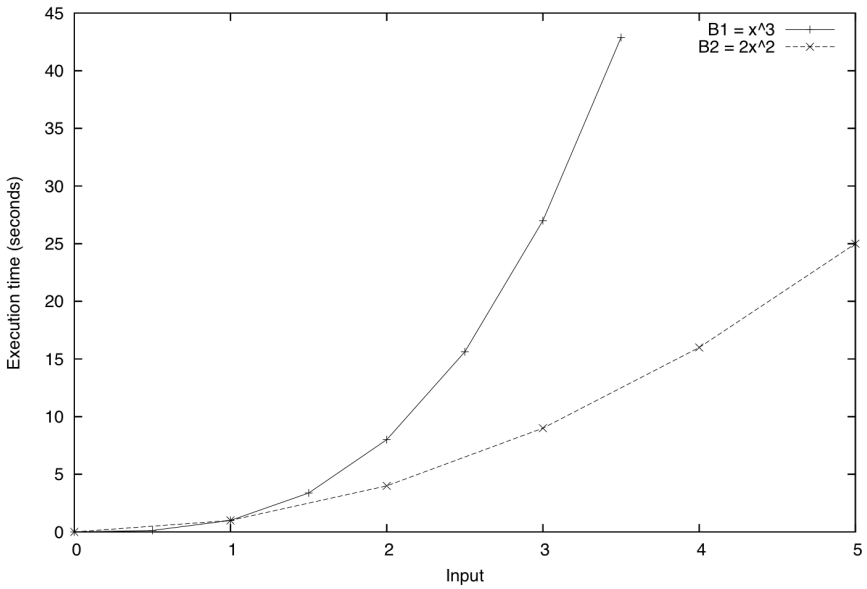


Figure 6. The performance models for both implementations of component B are plotted.

values of $x > 2$, the ideal implementation is the realization executing the lower-order performance model. This attribute enables the testing of the implementation selection library through modification of the inputs.

To demonstrate the measurement phase, the components were instrumented using the proxy-based measurement infrastructure described in this paper. Several runs were conducted with various sets of input values. For each set of input values, the results were checked by evaluating the models by hand to ensure the validity of the measurement framework.

The primary goal of the experiment was to examine the results produced by the optimization phase. The example resulted in the component call-graph depicted in Figure 7. Note that although this call-graph resembles the component wiring diagram, this will not always be the case. Components will only appear once in a wiring diagram, but may appear multiple times in a call-graph, dependent on how many distinct components invoke their methods. Threshold values of 10% were used for α and β. As expected, the pruner identified components A and B (along with the Driver) as the core components and pruned components C and D. Using the performance models for the components, the optimization library was utilized to select the optimal configuration. At this stage, the performance models were 'hard coded' into the optimization phase. Model representation is identified as an area of future work.

For the experiment with input values of $x < 2$, implementations A2 and B1 were correctly identified as the best choices. With the input restricted to $x > 2$, components A1 and B2 were targeted as the optimal choices. In the runs where x was varied, the choice of implementations varied also. In each instance, the optimization library correctly identified the correct choices.

DOI: 10.1002/cpe

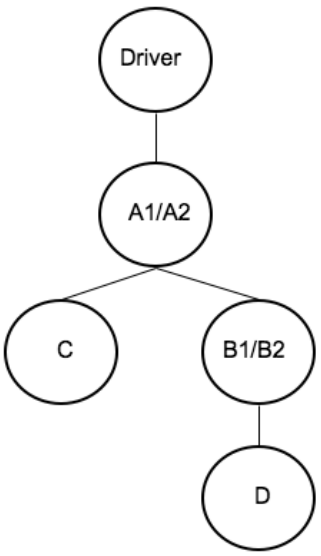


Figure 7. The call-graph produced by the component application pictured in Figure 4.

In addition, the measurement infrastructure was used in modeling the performance of a real-life scientific simulation code, the details of which may be found in [21].

6. CONCLUSIONS

This paper outlines a proxy-based software infrastructure designed to non-intrusively measure performance in a high-performance computing environment. The application performance model must take into account the performance of the individual components, as well as the interaction between components. In addition, a simple algorithm to identify the dominant components (from a performance point of view) in a component assembly was presented. This measurement framework and optimization library were demonstrated and verified on several example applications.

There are numerous opportunities for further investigation in the measurement framework and optimization library described in this paper. In the area of performance modeling, the question of model representation was not addressed. Currently, the optimization library uses models that have been 'hard coded'. A more user-friendly approach would allow the users to specify the models in a format that could be parsed and constructed at runtime by the library, possibly XML or MathML. Also, the performance models should incorporate a quality-of-service aspect. In the case study presented in [21], there exists a choice between two solver components. The solver that offers the best performance characteristics (i.e. faster time-to-solution and less variability in execution time) is often not the solver

DOI: 10.1002/cpe

chosen by scientists because it is not as accurate. Ideally, this type of quality-of-service data should be associated with the performance models.

Another area for future investigation is dynamic implementation selection. Preliminary work has begun on a *Optimizer* component that has the ability to swap one component for another at runtime, including inserting a proxy for the new component and updating all the necessary port connections. Future work would enable the *Optimizer* to analyze the actual performance of current components compared to expected performance. Any components that are not performing adequately and have an alternate implementation available could be replaced.

ACKNOWLEDGEMENTS

This research is supported at the University of Oregon by the U.S. Department of Energy, Office of Science, under contracts DE-FG03-01ER25501 and DE-FG02-05ER25680.

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

REFERENCES

1. Armstrong R, Gannon D, Geist A, Keahy K, Kohn S, McInnes L, Parker S, Smolenski B. Towards a Common Component Architecture for high performance scientific computing. *Proceedings of the 8th International Symposium on High Performance Distributed Computing 1999*, Redondo Beach, CA, 3–6 August 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999.
2. Allan B, Armstrong R, Wolfe A, Ray J, Bernholdt D, Kohl J. The CCA core specifications in a distributed memory SPMD framework. *Concurrency: Practice and Experience* 2002; **14**(5):323–345.
3. Lefantzi S, Ray J, Najm H. Using the Common Component Architecture to design high performance scientific simulation codes. *Proceedings of the International Parallel and Distributed Processing Symposium 2003*, Nice, France, 22–26 April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.
4. Lefantzi S, Ray J. A component-based scientific toolkit for reacting flows. *Proceedings of the 2nd MIT Conference on Computational Fluid and Solid Mechanics 2003*, Boston, MA, 17–20 June 2003. Elsevier: New York, 2003.
5. Ray J, Kennedy C, Lefantzi S, Najm H. High-order spatial discretizations and extended stability methods for reacting flows on structured adaptively refined meshes. *Third Joint Meeting of the U.S. Sections of The Combustion Institute*, Chicago, IL, March 2003 (CD-ROM).
6. Lefantzi S, Kennedy C, Ray J, Najm H. A study of the effect of higher order spatial discretizations in SAMR (Structured Adaptive Mesh Refinement) simulations. *Proceedings of the Fall Meeting of the Western States Section of the Combustion Institute*, Los Angeles, CA, October 2003 (CD-ROM).
7. Object Management Group page. http://www.omg.org/ [9 August 2005].
8. Maloney J. *Distributed COM Application Development using Visual C++ 6.0*. Prentice-Hall: Englewood Cliffs, NJ, 1999.
9. Englander R. *Developing Java Beans (Java Series)*. O'Reilly and Associates: Sebastopol, CA, 1997.
10. Mos A, Murphy J. Performance monitoring of Java component-oriented distributed applications. *Proceedings of IEEE 9th International Conference on Software Telecommunications and Computer Networks*, Split, Croatia, 9–13 October 2001. FESB-Split: Croatia, 2001.
11. Sridharan B, Dasarathy B, Mathur A. On building non-intrusive performance instrumentation blocks for CORBA-based distributed systems. *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium*, Chicago, IL, 27–29 March 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.
12. Sridharan B, Mundkur S, Mathur A. Non-intrusive testing, monitoring and control of distributed CORBA objects. *Proceedings of TOOLS Europe 2000*, St. Malo, France, June 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.
13. Ribler R, Simitci H, Reed D. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems (Performance Data Mining Special Issue)* 2001; **18**(1):175–187.

DOI: 10.1002/cpe

14. Ribler R, Vette J, Simitci H. Autopilot: adaptive control of distributed applications. *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing 1998*, Chicago, IL, 28–31 July 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998.
15. Tapus C, Chung I, Hollingsworth J. Active harmony: Towards automated performance tuning. *Proceedings of Supercomputing 2002*, Baltimore, MD, 16–22 November 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002.
16. Liu H, Parashar M. Enabling self-management of component-based high-performance scientific applications. *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, Research Triangle Park, NC, July 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005.
17. Norris B, Veljkovic I. Performance monitoring and analysis components in adaptive PDE-based simulations. *Submitted to the 24th International Symposium on Computer Performance, Modeling, Measurements and Evaluation*, Juan-les-Pins, France, October 2005. *Argonne National Laboratory Preprint* ANL/MCS-P1221-0105.
18. Malony A, Shende S. Performance technology for complex parallel and distributed systems. *In Distributed and Parallel Systems: From Concepts to Applications*, Kotsis G, Kacsuk P (eds.). Kluwer: Boston, MA, 2000; 37–46.
19. TAU: Tuning and Analysis Utilities page. http://www.cs.uoregon.edu/research/tau/ [9 August 2005].
20. Huck K, Malony A, Bell R, Li L, Morris A. Design and implementation of a parallel performance data management framework. *Proceedings of the International Conference on Parallel Processing*, Oslo, Norway, June 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005.
21. Ray J, Trebon N, Shende S, Armstrong R, Malony A. Performance measurement and modeling of component applications in a high performance computing environment: A case study. *Proceedings of the International Parallel and Distributed Processing Symposium 2004*, Santa Fe, NM, 26–30 April 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004.
22. PDT: Program Database Toolkit page. http://www.cs.uoregon.edu/research/pdt/ [9 August 2005].

DOI: 10.1002/cpe