Modular, Fine-Grained Adaptation of Parallel Programs

Pilsung Kang[1], Naresh K. C. Selvarasu[2], Naren Ramakrishnan[1], Calvin J. Ribbens[1], Danesh K. Tafti[2], and Srinidhi Varadarajan[1]

[1] Department of Computer Science, Virginia Tech, Blacksburg, VA 24061
[2] Department of Mechanical Engineering, Virginia Tech, Blacksburg, VA 24061

Abstract. We present a modular approach to realizing fine-grained adaptation of program behavior in a parallel environment. Using a compositional framework based on function call interception and manipulation, the adaptive logic to monitor internal program states and control the behavior of program modules is written and managed as a separate code, thus supporting centralized design of complex adaptation strategies for adapting to dynamic changes within an application. By 'catching' the functions that execute in synchronization across the parallel environment and inserting the adaptive logic operations at the intercepted control points, the proposed method provides a convenient way of synchronous adaptation without disturbing the parallel execution and communication structure already established in the original program. Applying our method to a CFD (computational fluid dynamics) simulation program to implement example adaptation scenarios, we demonstrate how effectively applications can change their behavior through fine-grained control.

1 Introduction

Implementing adaptive execution of an application in a distributed or parallel environment has been of much interest in recent years. The approaches to support program adaptation include: languages and compilers for specifying adaptation strategies [1, 2, 3] and runtime platforms or middleware for adaptive execution [4, 5, 6, 7]. These efforts are primarily centered around resource management to achieve efficient utilization of the environment, such as adaptive load-balancing and scheduling of application tasks, to match resource constraints or dynamic operating conditions of the environment. Adaptation schemes are 'coarse-grained' in these approaches in that cooperating processes of a distributed application are each abstracted as a task and adaptation strategies are designed to reassign the tasks onto the resources or to reorganize the execution flow among them. The metrics to initiate adaptation are usually based on measured history or estimates of the application execution time, which is a function of the environment's operating conditions such as available resources (e.g., number of processors) or physical characteristics of the resources (e.g., network bandwidth).

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 269–279, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Even with such support, however, adaptation schemes where functional behavior needs to be adjusted in response to internal changes to program state can be hard to design. Key challenges include the need for specifying adaptive parallel control points and monitoring state changes within the program, the need for executing parallel adaptation without disturbing the original execution flow, and the lack of support for centralizing adaptive logic operations in a separate module, thereby providing a compositional approach to dealing with the increased complexity of parallel adaptive applications. On the whole, coarse-grained approaches such as supported by runtime systems do not provide mechanisms to access fine-grained aspects of program state or to manipulate fine-grained behavior of the processes of a parallel application.

To address the issues, this paper presents a modular method for implementing fine-grained adaptive behavior with parallel programs using a function call interception (FCI) framework called *Invoke* [8]. Our work makes the following contributions:

- Factoring out the adaptation logic: A new code that implements the intended adaptive logic is written in a separate module and inserted by intercepting the functions of interest in a running application. This enables adaptation without code modification. We specifically target MPI programs written in the SPMD (Single Program, Multiple Data) style.
- Fine-grained control: Adaptation of program behavior such as simulation parameter adjustment or algorithm switching can be initiated in response to changes in internal computation states.
- Synchronous adaptation: By 'catching' global computation functions in the original program and plugging in new codes at the intercepted places, adaptive operations can be safely carried out without disturbing the parallel execution structure already established in the original program.

2 Compositional Approach for Parallel Adaptation

Invoke is a composition framework with a set of FCI APIs, through which every call to a function of interest is intercepted and program control is diverted to an associated *handler*, a piece of newly inserted code responsible for monitoring and modifying the target function's behavior. By specifying a target function to be manipulated by Invoke, we essentially define an adaptive control point over the original program, where newly developed modules can be introduced to maneuver the program toward the intended adaptive behavior. Thus, as Fig. 1 shows, composition through Invoke enables one to separately reason about application-specific adaptive strategies, factor them out in a centralized code, and plug in the adaptation code at control points to build an adaptive application.

2.1 Fine-Grained Program Adaptation

By defining adaptive control points at the interfaces of subprogram modules, the compositional approach conveniently achieves effective, fine-grained control over

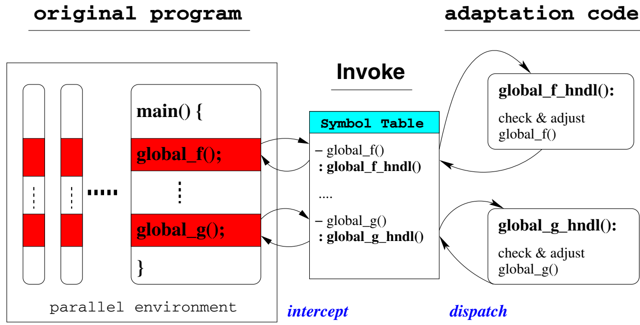


Fig. 1. Composition of an adaptive parallel application using Invoke

application behavior, where adaptation strategies can be designed to monitor and react to changes in internal program states. Global state variables can be accessed from the adaptivity code by declaring these variables as external. The Invoke framework also provides function parameter control APIs, which enables an extra level of flexibility in fine-grained adaptation. Function arguments are usually not exposed as globals in a program but still can hold important runtime program state for certain adaptation purposes. Through the parameter control APIs, dynamic program states that are communicated between modules can not only be accessed to check the computational progress, but also be manipulated to adjust the program's runtime behavior. We had previously presented such adaptation for sequential environments in [9] but here we focus on parallel execution environments.

2.2 Synchronous Parallel Adaptation

Implementing parallel adaptive behavior through the existing Invoke compositional framework requires adaptive logic operations to take place synchronously at clearly defined program control points that are shared across all the participating processes. This is important for implementing fine-grained adaptation strategies with SPMD programs where program behavior needs to change dynamically in response to changes in program state, because asynchronous adaptation in a parallel program can cause race conditions among the processes and make the entire computation invalid. For example, if one process changes a global simulation parameter or algorithm, and continues the computation, before another process makes the corresponding adaptation, the result may be inconsistent. Therefore, a synchronous adaptation mechanism is essential for implementing fine-grained adaptation in a parallel environment, where program behavior (typically in response to changes in internal program states) needs to adapt dynamically.

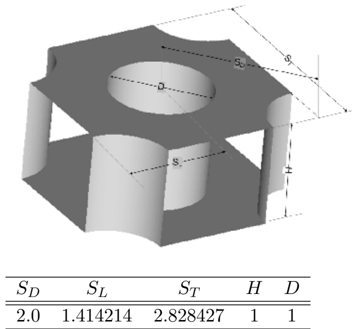
Synchronous adaptation can degrade performance if the adaptivity code involves extra global communication and synchronization. To mitigate the potential performance slowdown caused by adaptive global operations, we plug in the adaptivity code at global synchronization points that are already established

in the original program, thus placing separate barriers (one from the original code and the other from the new adaptivity code) close together and making the combined overhead smaller. By having the adaptivity operations "piggyback" onto the existing communications that are executed synchronously across the parallel environment, monitoring and adjusting the program states can also be performed synchronously without explicitly using extra global operations.

3 Adaptive CFD Simulations

In this section, we apply our framework to the GenIDLEST CFD simulation code [10] to automatically adjust the simulation time step value and dynamically change the flow model. Written in Fortran 90 with MPI to simulate CFD problems, GenIDLEST solves the time-dependent incompressible Navier-Stokes and energy or temperature equations.

The simulated problem is a pin fin array geometry as shown in Fig. 2. Extended surfaces or fins have been used extensively to augment the heat/mass transfer from or to a surface primarily by increasing the transfer area and/or increasing the heat/mass transfer coefficient. Reducing the size and weight requirements of equipment necessitates the need for optimal designs of these systems, which in turn requires a detailed understanding of flow and heat transfer characteristics. The schematic and the geometric parameters of the pin fin array under consideration, along with the dimensions of interest, are listed in Fig. 2. The slenderness ratio is set to 1. For the GenIDLEST simulation, we divided the geometry into 16 block structures so that the maximum degree of parallelism is 16, where each block is assigned to one MPI process.



S_D	S_L	S_T	H	D
2.0	1.414214	2.828427	1	1

Fig. 2. Schematic and Geometric Parameters of Pin Fin Array under Consideration

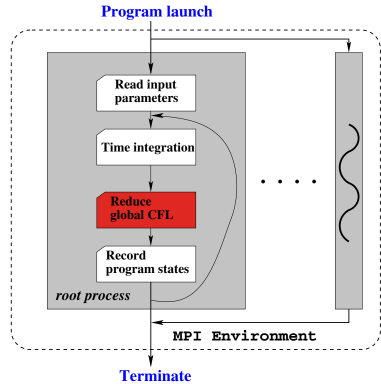


Fig. 3. GenIDLEST Execution Flow

3.1 Automatic Adjustment of Simulation Time Step

The stability of the simulation depends on the time step size used. Based on observed Courant-Friedrich-Levi (CFL) numbers one could discern if the simulation

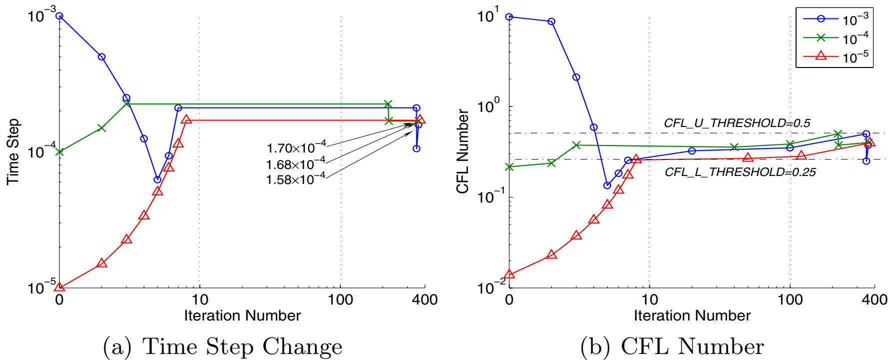
is proceeding towards convergence or is becoming unstable. Current practice of running GenIDLEST simulations records intermediate results at the end of a preset number of iterations onto the disk, thereby allowing the user to stop the execution and restart from the last known stable state when the user determines the running simulation is diverging. By plugging in a simple adaptivity module, the enhanced GenIDLEST simulation (requiring no modifications to the original GenIDLEST code) will incrementally adjust the time step value at runtime, allowing the computation to proceed in a stable manner.

Implementation: Fig. 3 shows the execution flow of the GenIDLEST simulation. At the end of every preset number of iterations, a local CFL number is calculated by each MPI process, and then the global CFL value is computed using a reduction operation (`mpi_allreduce`) across all the processes. This point is a good candidate for adaptivity code insertion, since by catching and imposing operations at this synchronization point, the newly inserted code can also be executed in synchronization, thereby avoiding dangerous race conditions among the processes. Furthermore, catching the global reduction call also makes it easy to monitor the global CFL number because its value is passed as the second parameter of the function. Invoke's parameter accessing APIs can be utilized to access this value. In the adaptive logic, we employed a simple multiplicative increase, multiplicative decrease algorithm with upper (`CFL_U_THRESHOLD`) and lower (`CFL_L_THRESHOLD`) threshold values for the CFL number, such that the time step is increased or decreased by a preset factor if the current CFL number becomes out of the bounds defined by the thresholds. Importantly, the entire adaptive logic operations are performed synchronously at the call sites of `mpi_allreduce` without involving any extra global operations, thereby achieving efficient parallel program adaptation. The implementation aspects of this and the following adaptation scenarios are summarized in Table 1.

Experimental Results: Fig. 4 shows the results of GenIDLEST enhanced with the constructed adaptivity module for the pin fin array simulation, with different initial values of time step ranging from 10^{-3} to 10^{-5}. `CFL_U_THRESHOLD` and `CFL_L_THRESHOLD` were set to 0.5 and 0.25, respectively. The graphs show how the CFL value changes as the time step parameter is controlled by the new

Table 1. Implementation Aspects of GenIDLEST Adaptation

	Change of Time Step	**Change of Flow Models**
Purpose	improve stability	enhance accuracy
Type of Scheme	automatic adjustment	user's dynamic decision
States to Monitor	CFL number communicated by `mpi_allreduce`	stream-wise velocity written to a log
Control Point	`mpi_allreduce` in `calc_cfl`	`calc_cfl` in time integration loop
Adaptive Logic	adjust time step to confine CFL number within certain bounds	switch flow model (`i_les`) and activate turbulent data structures
Communication	not necessary	broadcast of user's decision



(a) Time Step Change (b) CFL Number

Fig. 4. Automatic Adjustment of the Time Step Parameter

module, thereby maintaining the stability of the simulation. Interestingly, it also shows that the time step in all cases converge to somewhere around 1.7×10^{-4}, which might be the optimal value for the model, regardless of different starting values. Therefore, an adaptive logic based on a sophisticated CFD theory might be devised to find the optimal time step for more generalized problems through our composition method.

3.2 Runtime Change of Flow Models

The predicted heat transfer and flow characteristics depend on the selection of the appropriate flow model. A fundamental distinction is between laminar and turbulent flow models, and simulations of interest often require a switch from one to the other. This problem becomes acute when the Reynolds number is in the transition region between laminar and turbulent flows. Thus it becomes important to change the flow model from laminar to turbulent once instabilities arise in the flow field, for a simulation that is started assuming the flow is laminar.

Two Large Eddy Simulation (LES) turbulent models are considered in this study – Smagorinsky model (SM) and dynamic Smagorinsky model (DSM) [11]. The most commonly used model is the Smagorinsky model, where the eddy viscosity of the subgrid scales is obtained by assuming that the energy production and destruction are in equilibrium. The drawback of this model is that the model coefficient is kept constant, while in reality it should vary within the flow field depending on the local state of turbulence. The dynamic Smagorinsky model computes the model coefficient dynamically, which overcomes the deficiencies of the Smagorinsky model by locally calculating the eddy viscosity coefficient to reflect closely the state of the flow [11]. The advantage of the DSM model is that the need to specify the model coefficient is eliminated, making the model more self-contained, but with an additional computational expense of 10-15%.

Implementation: The simulated flow model in GenIDLEST is set initially by the user through an input specification parameter, namely `i_les`: 0 for laminar,

1 for Smagorinsky, and 2 for Dynamic Smagorinsky model. Hence, the program state needs to be accessed and changed at runtime via this variable. Importantly, the change should be made synchronously across all processes to maintain the consistency of the parallel computation. To this end, we plug in the adaptivity module at the call site of the CFL reduction function as shown in Fig. 3, because it is executed in synchronization across all the MPI processes, providing a safe place for carrying out adaptation operations without modifying the original code and disturbing the parallel execution flow already established in the original GenIDLEST. Specifically, the adaptivity code checks if the user wants to change the flow model, for which we make use of Unix signals (e.g., `SIGUSR1`) that can handle immediate, unanticipated user decisions to switch the flow model. These user-sent signals set a flag in the root process, which will pause accordingly with a simple user interface in the next iteration to accept the user's adaptation decisions, which in turn are broadcast to the other processes.

Experimental Results: The variation of the velocity in the direction of flow (stream-wise) is plotted in Fig. 5, showing the points in time when the flow models are switched from laminar to SM and then to DSM. The stream-wise velocity initially decreases, as the simulation proceeds towards the solution, which occurs till about 0.6 time units. After this simulation time, we see that the stream-wise velocity tends to vary with time, indicating the development of flow instabilities, and implying that the initial assumption of laminar flow is no longer valid. Thus the model is switched to SM at time 1.0. The drawback with the SM model, as mentioned earlier, is that the model coefficient is set to a constant value, but in reality the coefficient varies with the local state of turbulence, thus it becomes imperative to change the model from SM to DSM. This switch is done after a few hundred iterations (at time 1.4) to make sure that the switch from laminar to turbulent model does not introduce instabilities in the computation. The switch from laminar to turbulent flow model has a significant effect on the heat transfer. This is shown in Fig. 6(a) and 6(b), which show the variation of the Nusselt number at the channel walls, which is a measure of heat transfer at that location. The dotted line shows the region of interest, which is at the front of the pin in the line of fluid flow. The laminar flow model does not capture the

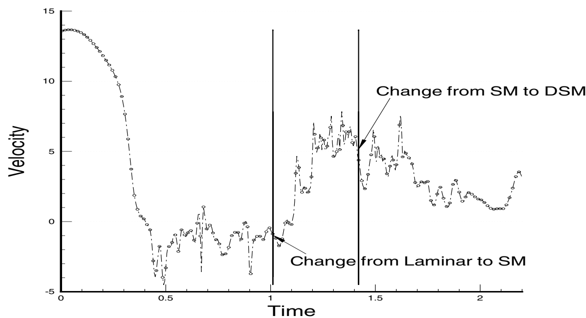
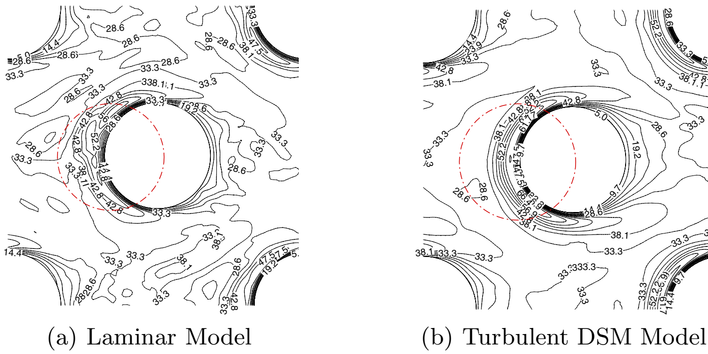


Fig. 5. Variation of Stream-Wise Velocity with Flow Model Change



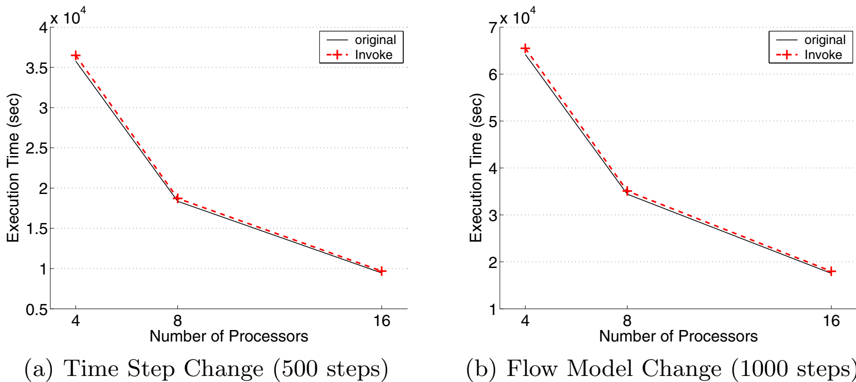
(a) Laminar Model (b) Turbulent DSM Model

Fig. 6. Dynamic Flow Model Change from Laminar to Turbulent in a CFD Simulation

heat transfer effects at the front of the pin, predicting lower heat transfer rates at the pin front than the turbulent model, thus justifying the model switch from laminar to DSM. This switch shows the capabilities of the adaptive scheme, since to effect the switch without it would have meant stopping the current execution and then restarting the simulation after effecting the required change.

4 Adaptation Overhead

The runtime overhead of our adaptation method comes from catching the function calls at adaptive control points, which in itself does not involve any global operations that cause communication overhead. The catching overhead is measured at 0.10μs per call on average on an AMD Opteron 240 1.4GHz dual-processor machine with 1GB memory, which translates to 140 CPU cycles. Since the catching cost is fixed, the relative overhead depends on the number of interceptions and the entire execution profile of an application. That is, the overhead increases as the number of adaptive control points increases. Still, the catching cost is relatively insignificant if the application spends most of its time on executing other parts of the computation than at control points.



(a) Time Step Change (500 steps) (b) Flow Model Change (1000 steps)

Fig. 7. Invoke Overhead with GenIDLEST Simulations

In the adaptive GenIDLEST simulations, control points are intercepted only once at the end of every preset number of iterations of the time integration loop, while most of the computing time is spent inside the loop. As a result, the catching overhead is negligible compared to the whole simulation profile. For example, Fig. 7 shows execution time of the GenIDLEST simulations where the Invoke framework is imposed at control points in the time step change and in the flow model change scenario, respectively, but with no adaptation operations. Across the 3 configurations with varying number of processors, the costs for catching 500 calls of `mpi_allreduce` during 500 time steps in the time step change example were measured to be less than 0.7% in all cases compared to the original GenIDLEST simulations (Fig. 7(a)). Similarly, the overhead is less than 0.95% for catching 1000 calls of `calc_cfl` during 1000 time steps in the flow model change example (Fig. 7(b)).

5 Related Work

In the language and compiler approaches for implementing program adaptation, our work is similar to Program Control Language (PCL) [2] in that centralized design of adaptation strategies can be specified at a high level for distributed programs. The expressive power of PCL comes from its underlying framework which offers a global representation of the distributed program as a graph of task nodes, the static task graph (STG), connected by edges indicating precedence relationships. Each adaptation primitive of PCL maps to a sequence of graph-changing operations on the STG of the target program. The Invoke framework provides more fine-grained control than PCL STGs by supporting monitoring and manipulating of state variables internal to a program.

In Grid and cluster computing, there is a large body of research work on runtime platforms for supporting program adaptation at the level of middleware or runtime platforms [4,5,6,7]. However, as their objective is to implement middleware support for adaptation between the application and the underlying execution layer, these efforts focus on resource management towards efficient utilization of the environment, such as load-balancing and scheduling of application tasks, where coarse-grained strategies based on resource constraints or external operating parameters are employed. In contrast, our work implements a parallel adaptation framework that can adjust fine-grained aspects of program state and behavior by monitoring dynamic progress of the computation itself.

Dynamic binary instrumentation tools such as DynInst [12] offer a modular, language-independent way of code modification, so that new code modules can be transparently combined with existing software. Since the accompanying overhead is significant while they perform code instrumentation at program runtime, they are usually developed for sophisticated programs analysis purposes [13] rather than as a tool to realize program behavior adaptation.

6 Conclusions

The proposed compositional framework offers a modular way of implementing fine-grained program adaptation in a parallel environment. By defining adaptive control points at the functions that execute in synchronization across the parallel environment, adaptive logic operations can safely be executed without interfering with the parallel execution structure of the original program. In future work, we intend to define 'adaptivity schemas' that abstract our recurring templates of adaptivity and that can be 'weaved' over an unmodified program, akin to aspect oriented programming. We also intend to explore more dynamic and less synchronous scenarios of parallel program adaptation.

References

1. Voss, M.J., Eigemann, R.: High-level Adaptive Program Optimization with ADAPT. In: PPoPP 2001: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pp. 93–102. ACM, New York (2001)
2. Ensink, B., Stanley, J., Adve, V.: Program Control Language: A Programming Language for Adaptive Distributed Applications. J. Parallel Distrib. Comput. 63(11), 1082–1104 (2003)
3. Du, W., Agrawal, G.: Language and Compiler Support for Adaptive Applications. In: SC 2004: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, Washington, DC, USA, pp. 29–29. IEEE Computer Society Press, Los Alamitos (2004)
4. Kennedy, K., Mazina, M., Mellor-Crummey, J.M., Cooper, K.D., Torczon, L., Berman, F., Chien, A.A., Dail, H., Sievert, O., Angulo, D., Foster, I.T., Aydt, R.A., Reed, D.A., Gannon, D., Johnsson, S.L., Kesselman, C., Dongarra, J., Vadhiyar, S.S., Wolski, R.: Toward a Framework for Preparing and Executing Adaptive Grid Programs. In: IPDPS 2002: Proceedings of the 16th International Parallel and Distributed Processing Symposium, Washington, DC, USA, pp. 171–175. IEEE Computer Society Press, Los Alamitos (2002)
5. Buaklee, D., Tracy, G.F., Vernon, M.K., Wright, S.J.: Near-Optimal Adaptive Control of a Large Grid Application. In: Proceedings of the 16th International Conference on Supercomputing, pp. 315–326. ACM Press, New York (2002)
6. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D.: Adaptive Computing on the Grid using AppLeS. IEEE Transactions on Parallel and Distributed Systems 14(4), 369–382 (2003)
7. Janjic, V., Hammond, K., Yang, Y.: Using Application Information to Drive Adaptive Grid Middleware Scheduling Decisions. In: Proceedings of the 2nd Workshop on Middleware-Application Interaction, pp. 7–12. ACM, New York (2008)
8. Heffner, M.A.: A Runtime Framework for Adaptive Compositional Modeling. Master's thesis, Blacksburg, VA, USA (2004)
9. Kang, P., Cao, Y., Ramakrishnan, N., Ribbens, C.J., Varadarajan, S.: Modular Implementation of Adaptive Decisions in Stochastic Simulations. In: SAC 2009: Proceedings of the 24th Annual ACM Symposium on Applied Computing, pp. 995–1001 (March 2009)

10. Tafti, D.: GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows. In: Proceedings of the ASME Fluids Engineering Division (FED), vol. 256, ASME-IMECE (November 2001)
11. Germano, M., Piomelli, U., Moin, P., Cabot, W.H.: A Dynamic Subgrid-Scale Eddy Viscosity Model. Physics of Fluids A: Fluid Dynamics 3(7), 1760–1765 (1991)
12. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. Int. J. High Perform. Comput. Appl. 14(4), 317–329 (2000)
13. Schulz, M., Ahn, D., Bernat, A., de Supinski, B.R., Ko, S.Y., Lee, G., Rountree, B.: Scalable Dynamic Binary Instrumentation for Blue Gene/L. SIGARCH Comput. Archit. News 33(5), 9–14 (2005)