Enabling Self-management of Component-based High-Performance Scientific Applications *

Hua Liu and Manish Parashar
The Applied Software Systems Laboratory
Dept of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ 08854, USA
Email:{marialiu, parashar}@caip.rutgers.edu

Abstract

Emerging high-performance parallel/distributed scientific applications and environments are increasingly large, dynamic and complex. As a result, it requires programming systems that enable the applications to detect and dynamically respond to changing requirements, state and execution context by adapting their computational behaviors and interactions. In this paper, we present such a programming system that extends the Common Component Architecture to enable self-management of component-based scientific applications. The programming system separates and categorizes operational requirements of scientific applications, and allows them to be specified and enforced at runtime through re-configuration, optimization and healing of individual components and the application. Two scientific simulations are used to illustrate the system and its self-managing behaviors. A performance evaluation is also presented.

1 Introduction

Emerging high-performance parallel/distributed simulations and the phenomena they model are large, complex, multi-phased/multi-scale, dynamic, and heterogeneous (in time, space, and state). These simulations implement various numerical algorithms, physical constitutive models, domain discretizations, domain partitioners, communication/interaction models, and a variety of data structures. Further, the choices of algorithms and models have performance implications which are typically not known a priori. Advanced adaptive solution techniques, such as variable step time integrators and adaptive mesh refinement, add a new dimension to the complexity - the application behaviors and its requirements change as the simulation proceeds. This dynamism, coupled with the complexity and dynamism of emerging parallel/distributed execution environments, poses a new set of application development and runtime management challenges. For example, component behaviors and their compositions can no longer be statically defined. Further, their performance characteristics can no longer be derived from a small synthetic run as they depend on the state of the simulations and the underlying system. Algorithms that worked well at the beginning of the simulation may become suboptimal as the solution deviates from the space the algorithm was optimized for or as the execution context changes.

Addressing the challenges outlined above requires that applications be capable of detecting and dynamically responding to changing requirements, state and execution context. In this paper we investigate self-managing high-performance simulations. We also present a prototype implementation and evaluation of a programming system for developing self-managing applications based on the DoE Common Component Architecture (CCA) and the Ccaffeine framework [7]. Finally, we present the self-managing shock hydrodynamics simulation and CH_4 ignition simulation as case studies. Specific contributions of this paper include: (1) extension of CCA to enable the definition of managed components and applications; (2) design and implementation of a runtime framework to support self-managing component and application behaviors using dynamically defined rules; (3) a three-phase rule execution model to enable consistent and efficient rule execution for distributed/parallel scientific applications; and (4) support for performance driven self-management using the TAU framework [4].

The rest of the paper is organized as follows. Section 2 introduces the Common Component Architecture (CCA), investigates the performance characteristics of CCA-based scientific applications, and discusses their implication on application management. Section 3 presents a framework

*The research presented in this paper is supported in part by the National Science Foundation through grants ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826.

for the formulation and execution of self-managing scientific simulations based on the Ccaffeine CCA framework. Section 4 presents the case studies and experimental evaluations. Section 5 discusses related work. Section 6 presents a conclusion.

2 Component-Based Distributed/Parallel Scientific Applications

2.1 The Common Component Architecture (CCA)

Component-based software architectures do address some of the key requirements of emerging high-performance parallel/distributed scientific applications. Specifically, the DoE Common Component Architecture (CCA) and its implementation, the Ccaffeine framework [7], have been successfully used by a number of applications [13, 14, 15]. CCA supports the provides-uses design pattern. Components *provide* functions and *use* other components' functions via *ports*. Components are peers and independently developed. Further, CCA employs the *Single Component Multiple Data (SCMD)* model, where all processing nodes execute the same program structure.

Ccaffeine [7], developed at Sandia National Laboratories, implements the CCA core specification and provides the fast and lightweight glue to integrate external and portable peer components into a SCMD style parallel application. Components are created and exist within the Ccaffeine framework. They register themselves and their ports with the framework and are dynamically loaded and connected. As a result, the Ccaffeine framework maintains complete knowledge about an application. Further, all the components on the same processor reside in the same address space and these components interact with each other using method calls. Component interaction across processors use MPI [5].

2.2 Behavior and Performance of Component-based Scientific Applications

The component-based programming approach not only reduces the burden of developing scientific applications, but also benefits their runtime management. With componentization [7], the behavior and performance of an application can be interpreted as a composition of individual components. For example, the composite performance of a component assembly is determined by the performance of the individual components and the efficiency of their interaction [21]. Therefore, management behaviors can be systematically enforced at two separate levels - intra-component and inter-component.

The execution of scientific applications typically consists of a series of computational phases. Between two successive phases, computations within components and communications between components are paused, and the components are reconfigured for the next phase. This pause between phases has been called a *quiet interval*. Runtime management is usually performed during these *quiet intervals* to ensure the integrity of the numerical computations. Changes made to components/applications during *a quiet interval* are automatically applied in the next computational phase.

Finally, in case of the Ccaffeine framework, due to the underlying SCMD model, connections between components can be made by directly passing ports (i.e., pointers to pure virtual interfaces), which incur negligible overheads [7]. As a result, the overall performance of an application can be simply viewed as a function of the performance of its constituent components. Further, in case of scientific applications, the performance of a component is dominated by the cache performance of its implementation and the cost of inter-processor communications [21]. Cache performance is defined by the degree of data locality in computation algorithms and is affected by the cache size and cache management strategies used by the execution environment. Inter-processor communication costs are defined by software and algorithmic strategies used by the implementation (e.g., combining communication steps, minimizing/combining global reductions and barriers, overlapping communications with computations, etc.), and are affected by factors such as load-balance and communication channel congestion (due to competing application or possibly malicious attacks).

3 Self-management of Component-based Scientific Applications

As mentioned above, addressing the challenges of emerging high-performance scientific applications requires a programming system that enables the specification of applications, which can detect and dynamically respond, during their execution to changes in both the execution environment and application state. This requirement suggests that: (1) Applications should be composed from discrete self-managing components, which incorporate separate specifications for all of functional, non-functional and interaction-coordination behaviors. (2) The specifications of computational (functional) behaviors, interaction and coordination behaviors and non-functional behaviors (e.g. performance, fault detection and recovery, etc.) should be separated so that their combinations are compose-able. (3) The interface definitions of these components should be separated from their implementations to enable heterogeneous components to interact and to enable dynamic selection of components.

Component-based scientific simulations and the CCA architecture address some of these requirements and support application maintainability and extensibility. The capability of dynamically swapping components has been incorporated into the CCA specification and implemented by the Ccaffeine framework. However, enabling self-managing components/applications requires extending CCA to enable components that can adapt their behaviors and interactions to their current state and execution context in an autonomic manner. In this section we describe an extension of the CCA architecture, and specifically the Ccaffeine framework [7], to support self-management. Using the approaches proposed in [16, 17, 18], this consists of extending CCA components (including legacy components) to support monitoring and control and extending the Ccaffeine framework to support consistent and efficient rule-based intra-component and inter-component self-management behaviors.

3.1 Defining Managed Components

In order to monitor and control the behaviors and performance of CCA components, the components must implement and export appropriate "sensor" and "actuator" interfaces. Note that the sensor and actuator interfaces are similar to those used in monitoring/steering systems [12, 22, 23]. However, these systems focus on interactive management through users manually invoking sensors/actuators, while this paper focuses on automatic management based on user-defined rules. Adding sensors requires modification/instrumentation of the component source code. In case of third-party and legacy components, where such a modification may not be possible or feasible, proxy components [21] are used to collect relevant component information. A proxy provides the same interfaces as the actual component and is interposed between the caller and callee components to monitor, for example, all the method invocations for the callee component. Actuators can similarly be implemented either as new methods that modify internal parameters and behaviors of a component, or defined in terms of existing methods if the component cannot be modified. The adaptability of the components may be limited in the latter case. In the CCA based implementation, both sensors and actuators are exposed via invoking the 'addSensor' or 'addActuator' methods defined by a specialized *RulePort*, which is shown in Figure 1.

Management and adaptation behaviors can be dynamically specified by developers in the form of rules. Two classes of rules are defined.

- *Component rules* address intra-component management. These rules manage the runtime behaviors of individual components, including dynamic selection of algorithms, implementations, data representation, input/output format used by the components, etc., based on the current state and execution context of the component.
- *Composition rules* address inter-component management. These rules manage the structure of the application and the interaction relationships among components based on the current application/system state, changing requirements, and changing execution context. Intra-component management behaviors include dynamic composition of components, definition of coordination relationships and selection of communication mechanisms. For example, composition rules can be used to add, delete or replace a component.

```
class RulePort: public virtual Port {
public:
    RulePort(): Port() { }
    virtual ~RulePort() { }
    virtual void loadRules(const char* fileName) throw(Exception) = 0;
    virtual void addSensor(Sensor *snr) throw(Exception) = 0;
    virtual void addActuator(Actuator *atr) throw(Exception) = 0;
    virtual void setFrequency() throw(Exception) = 0;
    virtual void fire() throw(Exception) = 0;
};
```

Figure 1. The *RulePort* specification.

Management rules incorporate high-level guidance and practical human knowledge in the form of conditional if-then expressions, i.e., IF *condition* THEN *action*. This simple construction of rules is deliberately used to enable efficient execution and minimize impact on the performance of the application. The *condition* is a logical combination of sensors (exposed by components) and performance data, and the *action* consists of a sequence of invocations of actuators exposed by components. The rules are interpreted and executed by the runtime framework, which is discussed in the next section.

3.2 Enabling Runtime Self-management

To enable runtime self-management, two specialized component types are defined (see Figures 2 and 3): (1) Component manager that monitors and manages the behaviors of individual components, e.g., selecting the optimal algorithms or modifying internal states, and (2) Composition manager that manages, adapts and optimizes the execution of an application at runtime. Both, component and composition managers are peers of user components and other system components, providing and/or using ports that are connected to other ports by the Ccaffeine framework. The two managers are not part of the Ccaffeine framework, and consequently provide the programmers the flexibility to integrate them into their applications only as needed.

The design of the component manager and composition manager components are based on the following observations and considerations.

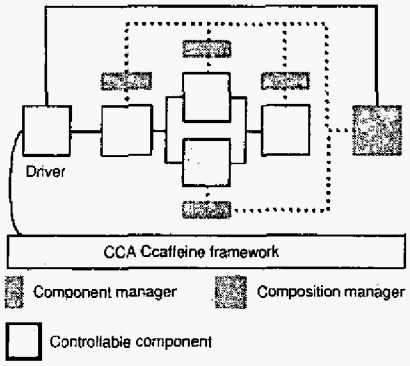

Figure 2. A self-managing application composed of 5 components. The black lines denote computational port connections between components, and the dotted lines are port connections constructing the management framework.

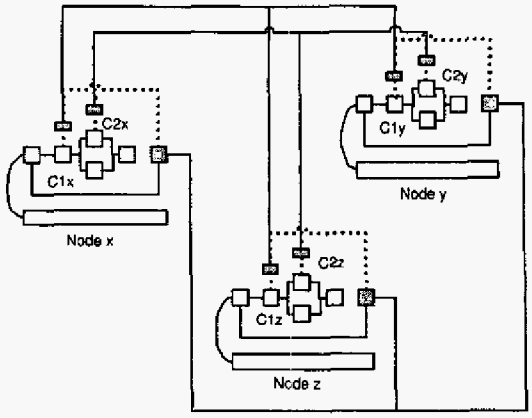

Figure 3. Distributed self-managing application shown in Figure 2 executed on three nodes. The black lines across nodes denote the interactions among manager instances. The dotted lines are port connections constructing the management framework within one node.

- Scientific applications may contain tens of components, but only a few of them need to be dynamically monitored and controlled. Therefore, we encapsulate the manager functionalities into two component types and provide programmers with the flexibility of integrating them with other components in the applications. For example, in Figure 3, only component $C1$ and $C2$ are associated with component managers for dynamic management.
- The manager functionalities are provided by components instead of being integrated within the Ccaffeine framework. This prevents the framework from being 'overweight' and thus avoids the resulting performance and maintenance implications.
- By encapsulating the manager functionality into these components and providing abstract interfaces for invoking this functionality, we can modify and improve the manager functionality without affecting other components and the framework. We can either add additional functionality into the manager components, or create other components that deal with specific management functions and integrate them with the manager components via the 'uses-provides design pattern' [7].

3.2.1 The Component Manager

Component managers provide the *RulePort* shown in Figure 1. They are instantiated only after the other application components are composed together. Their instantiation consists of two steps: first, instances of managed components expose their sensors and actuators to their respective component manager instances by invoking the 'addSensor' and 'addActuator' methods, and second, component rules are then loaded into component manager instances, possibly from disk files, by invoking the 'loadRules' method. This initialization of component manager instances is a one-time operation.

Management operations are performed during application *quiet intervals*. The managed components (or their proxies) invoke the 'fire' method of the *RulePort* to inform the component managers that they have entered into a quiet interval. This behavior must be explicitly programmed, possibly at the beginning/end of a computation phase or once every few phases, to establish the self-management frequency. Adaptations made during a quiet interval will be applied during the next computation phase.

3.2.2 The Composition Manager

The composition manager also provides the *RulePort* (shown in Figure 1). Composition manager instances are initialized by the CCA driver component to load in composition rules (possibly from a disk file) using the 'loadRules' method. These rules are then decomposed into sub rules, and delegated to corresponding component managers. The driver component notifies composition manager instances of quiet intervals by invoking the 'fire' method. During execution of the composition rules, composition manager instances collect results of sub rule execution from component manager instances, evaluate the combined rule, and notify component managers of actions to be performed. Possible

actions include adding, deleting, or replacing components. When replacing a managed component, the new component does not have to provide and use the exact same ports as the old one. However, the new component must at least provide all the active ports (those used by other components in the application) that are provided by the old component.

3.2.3 Rule Execution Model

A three-phase rule execution model [18] is used by the component managers to ensure consistent and efficient parallel rule execution. The three phases of rule execution are (1) batch condition inquiry, (2) condition evaluation and conflict resolution and reconciliation, and (3) batch action invocation.

During the batch condition inquiry phase, each component manager queries in parallel all the sensors used by the rules, gets their current values, and then generates the *pre-condition*. During the next phase, condition evaluation for all the rules is performed in parallel. And rule conflicts are detected at runtime when rule execution will change the *pre-condition* (defined as sensor-actuator conflicts), or the same actuator will be invoked with different values (defined as actuator-actuator conflicts). Sensor-actuator conflicts are resolved via disabling those rules that will change the *pre-condition*. Actuator-actuator conflicts are resolved through relaxing the pre-condition according to user-defined strategies until no actuator will be invoked with different values.

For example, consider component $C1$ with 3 algorithms: algorithm 1 has better cache performance but consumes a large communication bandwidth, algorithm 2 has comparatively more cache misses but only consumes a small bandwidth, and algorithm 3 demonstrates an acceptable cache miss and communication delay but has lower precision. It is possible that under certain conditions, rule evaluation may results in the selection of algorithm 1 and 2 at the same time to simultaneously decrease cache misses and communication delay, and maintain high-precision computation. This conflict is detected and resolved by relaxing the high-precision requirement, and therefore algorithm 3 can be selected. Further, the framework also provides mechanisms for reconciliation [18] among manager instances, which is required to ensure consistent adaptations in parallel SCMD applications, since each processing node may independently proposes different adaptation behaviors based on its local state and execution context.

The reconciliation for component rules consists of identifying and propagating the actions proposed by a majority of the nodes. If a majority is not found, an error is reported to the user. Composition rules are statically assigned one of the two priorities. A high priority means that the re-composition is necessary, while a low priority means the re-composition is optional. For the actions associated with composition rules with high priority are propagated to all the nodes. If there are multiple high priority rules with collisions, a runtime error is generated and reported to the user. The actions associated with composition rules with low priority, a cost model is used to approximate the performance gain of each action set and the action set with the best overall gain is selected and applied by all the nodes.

After conflict resolution and reconciliation, the *post-condition*, consisting of a set of actuators and their new values, is generated. And then during the batch action invocation phase, the actuators are actually set to the values in parallel.

Note that the rule execution model presented here focuses on correct and efficient execution of rules and providing mechanisms to detect and resolve conflicts at runtime. However, correctness of rules and conflict resolution strategies are responsibilities of the users.

3.3 Supporting Performance-driven Self-management

The TAU [4] framework is used for monitoring the performance of components and applications, and supporting performance-driven self-management. TAU can record inclusive and exclusive wall-clock time, process virtual time, hardware performance metrics such as data cache misses and floating point instructions executed, as well as a combination of multiple performance metrics, and help track application and runtime system level atomic events. Further, TAU is integrated with external libraries such as PAPI [2] or PCL [3] to access low-level processor-specific hardware performance metrics and low latency timers.

In our framework, TAU APIs are directly instrumented into the computational components, or into proxies in case of third-party and legacy computational components, and performance data is exported as sensors to component managers. Optimizations are used to reduce the overheads of performance monitoring. For example, as the cache-hit rate will not change unless a different algorithm is used or the component is migrated to another system with a different cache size and/or cache policies, monitoring of cache-hit rate can be deactivated after the first a few iterations and only re-activating when an algorithm is switched or the component is migrated. Similarly, inter-processor communication time is measured per message by default but this can be modified using the 'setFrequency' method in the *RulePort* to reduce overheads. Another possibility is to restrict monitoring to only those components that significantly contribute to the application performance. Composition managers can identify these components at runtime using mechanisms similar to those proposed in [26] and enable or disable monitoring as required. Finally, in case of homogeneous execution environments only a subset of

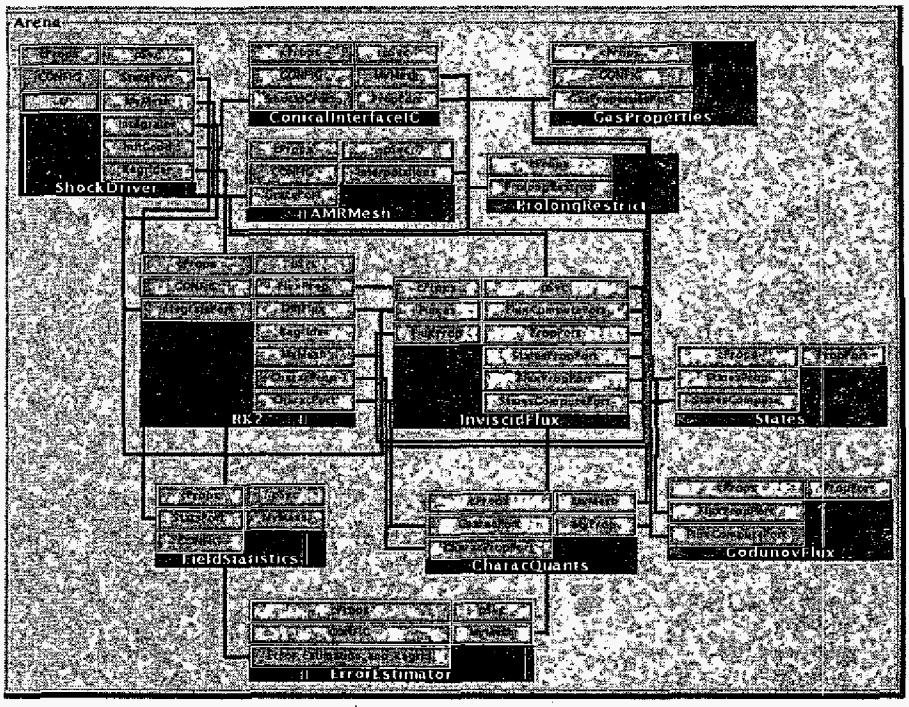

Figure 4. "Wiring" diagram of the shock-hydrodynamics simulation. A second-order Runge-Kutta (RK2)integrator drives InviscidFlux component – transformation into left and right (primitive) states is done by States and the Riemann problem solved by GodunovFlux. Sundry other components for determining characteristics' speeds (u + a, u - a, u), cell-centered interpolations etc. complete the code.

nodes may be monitored.

4 Case Studies

The operation of the programming system presented in this paper is illustrated using two applications, (1) a self-managing hydrodynamics shock simulation and (2) a self-managing CH_4 ignition simulation. An experimental evaluation of the programming system on a 64 node beowulf cluster is also presented. The cluster contains 64 Linux-based computers connected by 100 Mbps full-duplex switches. Each node has an Intel(R) Pentium-4 1.70GHz CPU with 512MB RAM and is running Linux 2.4.20-8 (kernel version).

4.1 A Self-Managing Hydrodynamics Shock Simulation

This application simulates the interaction of a hydrodynamic shock with a density-stratified interface. The system is modelled using the 2D Euler equation (inviscid Navier-Stokes). Details of the equations used and the interaction are presented in [20, 24, 25]. Figure 4 shows the assembly of components for the CCA-based implementation of the simulation. The simulation uses structured adaptive mesh refinement. In this implementation, the Runge-Kutta time integrator (**RK2**) with an **InviscidFlux** component supplies the right-hand-side of the equation on a patch-by-patch basis. This component uses a **ConstructLRStates** component to set up a Riemann problem at each cell interface, which is then passed to **GodunovFlux** for the Riemann solution. A **ConicalInterfaceIC** component sets up the problem - a shock tube with Air and Freon (density ratio 3) separated by an oblique interface that is ruptured by a Mach 10.0 shock. The shock tube has reflecting boundary conditions above and below and outflow on the right. The **AMRMesh** and **GodunovFlux** are the significant components in this simulation from the performance point of view, and is used to illustrate self-managing behaviors in the discussion below.

4.1.1 Scenario 1: Self-optimization via component replacement

An EFM algorithm, which is based on a gas-kinetic scheme [19], may be used instead of the Godunov method with **RK2** in the implementation described above. **Go-**

dunovFlux and **EFMFlux** demonstrate different performance behaviors and mean execution times as the size of the input array size increases, as shown in Figure 5. This difference in performance is primarily due to the difference in data locality and cache behaviors for the two implementations. **GodunovFlux** is more expensive than **EFMFlux** for large input arrays.

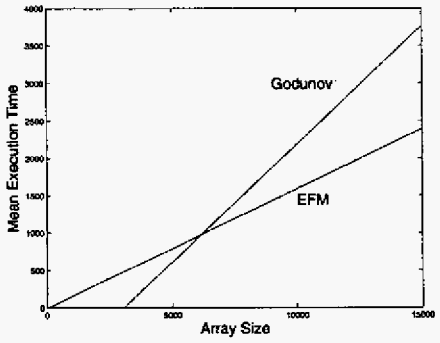

Figure 5. Average execution time for EFMFlux and GodunovFlux as a function of the array size (machine effects have be averaged out).

The appropriate choice of algorithm (Godunov or EFM) depends on simulation parameters, its runtime behaviors and the cache performance of the execution environment, and is not known a priori. In this scenario we use information about cache misses for **GodunovFlux** obtained using TAU/PCL/PAPI, to trigger self-optimization, so that when cache misses increase above a certain threshold, the corresponding instance of **GodunovFlux** is replaced with an instance of **EFMFlux**.

To enable the component replacement, one component manager is connected to **GodunovFlux** through the *RulePort* to collect performance data, evaluate rules, and perform runtime replacement. The component manager (1) locates and instantiates **EFMFlux** from the component repository, (2) detects all the provides and uses ports of **GodunovFlux**, as well as all the components connected to it, (3) disconnects **GodunovFlux** and delete all the rules related to **GodunovFlux**, (4) connects **EFMFlux** to related components and load in new rules, and finally (5) destroys **GodunovFlux**. The replacement is performed at a *quiet interval*. From the next calculation step, **EFMFlux** is used instead of **GodunovFlux**. However, other components in the application do not have to be aware of the replacement, since the abstract interfaces (ports) remain the same. After replacement, the cache behavior improves as seen in Figure 6.

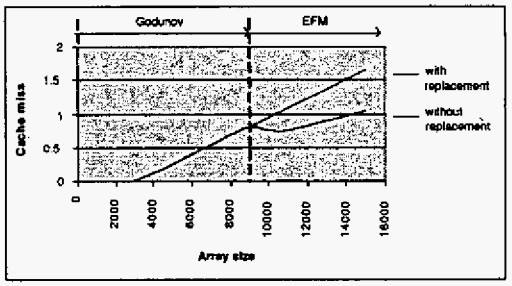

Figure 6. Replacement of GodunovFlux with EFMFlux to decrease cache misses.

4.1.2 Scenario 2: Self-optimization via component adaptation

The **AMRMesh** component supports structured adaptive mesh-refinement and provides two communication mechanisms. The first exchanges messages on a patch by patch basis and results in a large number of relatively small messages. The second packs messages from multiple patches to the same processor and sends them as a single message, resulting in a small number of much larger messages. Depending on the current latency and available bandwidth, the component can be dynamically adapted to switch the communication mechanism used.

In this scenario, we use the current system communication performance to adapt the communication mechanism used. As PAPI [2], PCL [3], and TAU [4] do not directly measure network latency and bandwidth, this is indirectly computed using communication times and message sizes. **AMRMesh** exposes communication time and message size as sensors, which are used by the component manager to get the current bandwidth as follows:

$$bandwidth = \frac{commTime_1 - commTime_2}{msgSize_1 - msgSize_2} \quad (1)$$

Here, '$commTime_1$' and '$commTime_2$' represent the communication times for messages with sizes '$msgSize_1$' and '$msgSize_2$' respectively. When the bandwidth falls below a threshold, the communication mechanism switches to patch by patch messaging (i.e., algorithm 1). This is illustrated in Figure 7. The algorithm switching happens at iteration 9 when channel congestion is detected, and results in comparatively smaller communication times in the following iterations.

4.1.3 Scenario 3: Self-healing via component replacement

While Godunov methods with **RK2** tend to be more accurate, they become unstable for stronger shocks and larger density ratios. One solution is to replace **GodunovFlux** in

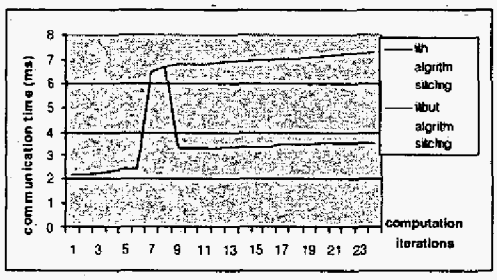

Figure 7. Dynamically switch algorithms in AMRMesh.

these cases with **EFMFlux**. The appropriate choice of algorithm (Godunov or EFMFlux) depends on the Mach number and the density ratio, and is once again not known a priori. In the best of cases, an algorithm will operate for some time before failing to converge and indicating an error; at other times, it will work "reliably" and produce wrong (even qualitatively wrong) results. In the case where an error can be identified, we have the option of dynamically replacing one algorithm by another by simply replacing the component implementing the algorithm. Of course, the same change has to be performed on all the processors. While dynamically changing components does raise some fundamental issues (e.g. in this case, the simulation is neither purely EFM-based nor Godunov-based, and is not mathematically consistent either), it is expected that the results will be at least qualitatively correct. Since such simulations often require substantial computational resources, obtaining qualitative answers may be preferable to simply exiting with an error.

In this scenario we investigate the dynamic replacement of **GodunovFlux** with **EFMFlux** so that it continues to provide qualitatively correct results. The adaptation is triggered when **GodunovFlux** fails to converge, i.e., its iteration count increases above a certain threshold, and causes the instance of component **GodunovFlux** to be replaced by an instance of component **EFMFlux**. The replacement process is the same as that described in scenario 1 above.

4.2 A Self-Managing CH_4 Ignition Simulation

This section focuses on the overall performance improvement of the CH_4 ignition simulation. The ignition process is represented by a set of chemical reactions, which appear and disappear when the fuel and oxidizer react and give rise to the various intermediate chemical species. In the simulation application, the chemical reactions are modelled as repeatedly solving the ChemicalRates equation (G) [1] with different initial conditions and parameters using one of a set of algorithms (called backward difference formula or BDF). The algorithms are numbered from 1 to 5, indicating the order of accuracy of the algorithm. BDF_5 is the highest order method, and is most accurate and robust. It may, however, not always be the quickest. As a result, the algorithm used for solving the equation G has to be selected based on current condition and parameters. In this application, the bulk of the time is spent in evaluating the equation G. Therefore, reducing the number of G evaluation is a sufficient indication of speed independent of experimental environments.

As shown in Figure 8, the rule-based execution decreases the number of invocation to equation G, and the percentage decrease is annotated for each temperature value. It results in an average 11.33% computational saving. As the problem becomes more complex (the computational cost of G increase), the computational saving will be more significant.

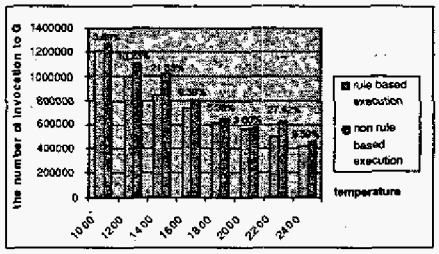

Figure 8. Comparison of rule based and non rule based execution of CH_4 ignition.

4.3 Experimental Evaluation

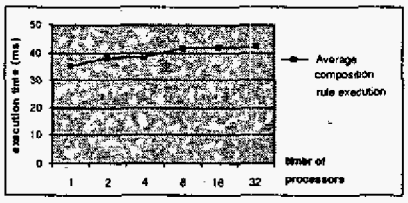

Figure 9. Overhead due to execution of composition rules.

An experimental evaluation of the overheads of the programming system is presented in this section. The first experiment evaluates the average execution time of a composition rule. The overhead of replacing **GodunovFlux** with **EFMFlux** is presented in Figure 9. The figure shows that, as the number of processors increases, the average execution time does increase but only slightly. This slight increase is primarily due to the time for reconciliation among

composition manager instances, which depends on the number of nodes involved. Once reconciliation is completed, component manager instances perform the replacement in parallel.

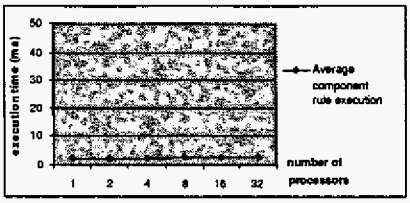

Figure 10. Overhead due to execution of component rules.

The second experiment evaluates the average execution of a component rule. The overhead of dynamically switching algorithms within the component **AMRMesh** is plotted in Figure 10. As seen from Figures 9 and 10, the average execution time of a composition rule is much larger than that of a component rule. This is because, in order to replace a component, the manager has to instantiate a new component, connect it to other components, and load new rules. However, the execution of component rules only involves invoking the component's actuators.

Note that while the framework does introduce overheads, the benefits of self-management would outweigh these overheads. Further, the overheads are not significant when compared to the typical execution time of scientific applications, which can be in hours, days, and even weeks.

5 Related Work

Related research efforts investigating systems for supporting dynamically adaptive applications can be classified based on the nature of the adaptations supported. In systems supporting *statically-defined adaptations*, adaptations must be defined at compile time and coded into the applications. These include systems that enable adaptations (1) by extending existing programming languages through providing templates (e.g., for adaptive scheduling as in [8]) or adaptation classes (e.g., to enable adaptive components as in [9]), or (2) by defining new adaptation languages (e.g., [11]). Systems in this category require that all possible adaptations must be known a priori. If new adaptations are required or application requirements change, the application code has to be modified and the application probably re-compiled.

In systems supporting *dynamically-defined adaptation*, adaptations (in the form of code, scripts or rules) can be added, removed and modified at runtime. The framework presented in this paper and system presented in [23, 28] fall into this category. These systems separate adaptation as an aspect and express it in terms of rules (conditions and actions) that can be dynamically managed. In [28], adaptations can only be performed at pre-defined method invocations, similar to 'injectors' and 'filters' [6]. Adaptation across multiple invocations are not supported. In the framework presented in this paper, rules are systematically composed of pre-defined sensors and actuators to provide more comprehensive adaptation behaviors. In this framework adaptations can occur at any quiet state rather than at pre-defined method invocations. Further, the framework differs from systems such as [23] in that it not only supports monitoring and steering within components but also enables management across components, e.g., by dynamically switching components.

ALua [27] is probably most closely related to the system presented in this paper. Both these systems separate configuration from computation and perform interaction, coordination and adaptation in an interpretive manner. Further, they both support the execution of dynamically defined adaptation behaviors (in the form of code, scripts or rules) to adapt application behaviors. However, the framework presented here uses components as the unit of adaptation, which allows more control of application consistency through encapsulation. The adaptation of individual components, such as setting the value of a variable or selecting an algorithm, are encapsulated within these components and access to them is controlled by constraints defined on the sensors and actuators. Similarly, the addition/deletion/replacement of components is restricted by their functional signatures and system requirements.

The performance-based self-management presented in this paper is also addressed in [10]. Adaptive behaviors such as algorithm selection and parameter adjustment presented in [10] are also supported in the framework presented here, both at the composition and the component levels. However, this framework differs from [10] in that adaptation behaviors are specified as rules that can be dynamically defined, rather that using hard-coded algorithms within the server.

6 Conclusion

This paper presented a programming system that enables self-managing component-based scientific applications capable of detecting and dynamically responding to changing requirements, state and execution context. The programming system extends the common component architecture (CCA) and the Ccaffeine framework. It enables the behaviors and interaction of components and applications to be defined using high level rules and provides a runtime framework for the correct and efficient execution of these rules. Mechanisms for detecting and resolving rule conflicts are provided. The operation of the programming system was il-

lustrated using a self-managing hydrodynamics shock simulation and a self-managing CH_4 ignition simulation. A performance evaluation was presented.

Current efforts include the investigation of additional scientific/engineering applications and additional adaptation behaviors, as well as deploying and evaluating the system on large HPC platforms such as DataStar, SDSC's IBM terascale machine.

7 Acknowledgement

The authors would like to acknowledge Jaideep Ray for all his help with the applications and self-managing scenarios, and Jaideep Ray, Benjamin A. Allan, and other members of the Ccaffeine team for making the Ccaffeine framework available to us. The authors would also like to acknowledge Sameer Shende for his help with the TAU framework. Part of the research presented in this paper was conducted by Hua Liu during her visit to Sandia National Laboratories in Summer 2004.

References

[1] GRI-Mech. http://www.me.berkeley.edu/gri_mech/.

[2] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/projects/papi.

[3] PCL - The Performance Counter Library. http://www.fz-juelich.de/zam/PCL.

[4] TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/paracomp/tau/tautools/.

[5] The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[6] M. Aksit and Z. Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *the 23rd International Conference on Distributed Computing Systems Workshops*, pages 84–89, Providence, Rhode Island, 2003. IEEE.

[7] B. A. Allan and et al. The CCA core specification in a distributed memory SPMD framework. *Concurrency Computation*, 14(5):323–345, 2002.

[8] F. Berman and et al. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[9] P. Boinot and et al. A declarative approach for designing and developing adaptive components. In *the 15th IEEE International Conference on Automated Software Engineering*, pages 111–119. IEEE, 2000.

[10] C. Ţăpuş and et al. Active harmony: Towards automated performance tuning. In *the IEEE/ACM SC2002 Conference*, pages 44–54, Baltimore, Maryland, 2002.

[11] G. Duzan and et al. Building adaptive distributed applications with middleware and aspects. In *the 3rd International Conference on Aspect-Oriented Software Development*, pages 66–73, Lancaster, UK, 2004. ACM.

[12] G. A. Geist and et al. Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. In *Environment and Tools for Parallel Scientific Computing Workshop*, Lyon, France, 1996.

[13] J. P. Kenny and et al. Component-Based Integration of Chemistry and Optimization Software. *J. Comput Chem*, 25:1717–1725, 2004.

[14] S. Lefantzi and et al. Using the common component architecture to design high performance scientific simulation codes. In *the International Parallel and Distributed Processing Symposium*, Nice, France, 2003.

[15] S. Lefantzi and et al. A component-based toolkit for reacting flows with high order spatial discretizations on structured adaptively refined meshes. *Progress in Computational Fluid Dynamics*, 2004. In press.

[16] H. Liu. A component-based programming framework for autonomic grid applications. Ph.D. proposal, 2004.

[17] H. Liu and et al. A component based programming framework for autonomic applications. In *the 1st IEEE International Conference on Autonomic Computing (ICAC-04)*, NYC, NY, USA, 2004.

[18] H. Liu and M. Parashar. A framework for rule-based autonomic management of parallel scientific applications. In *the 2nd IEEE International Conference on Autonomic Computing (ICAC-05)*, Seattle, Washington, 2005.

[19] D. I. Pullin. Direct Simulation Methods for Compressible Ideal Gas Flow. *J. Comp. Phys.*, 34:231–244, 1980.

[20] J. Ray and et al. Shock Interactions with Heavy Gaseous Elliptic Cylinders : Two Leeward-Side Shock Competition Models and a Heuristic Model for Interfacial Circulation Deposition at Early Times. *Phys. Fluids*, 12(3):707–716, 2000.

[21] J. Ray and et al. Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment : A Case Study. In *the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, USA, 2004.

[22] L. Renambot and et al. Cavestudy: an infrastructure for computational steering in virtual reality environments. In *the 9th IEEE International Symposium on High Performance Distributed Computing*, pages 57–61, Pittsburgh, PA, 2000.

[23] R. Ribler and et al. Autopilot: adaptive control of distributed applications. In *the High Performance Distributed Computing Conference*, pages 172–179, 1998.

[24] R. Samtaney and et al. Baroclinic Circulation Generation on Shock Accelerated Slow/Fast Gas Interfaces. *Phys. Fluids*, 10(5):1217–1230, 1998.

[25] R. Samtaney and N. Zabusky. Circulation Deposition on Shock-Accelerated Planar and Curved Density Stratified Interfaces : Models and Scaling laws. *J. Fluid Mech.*, 269:45–85, 1994.

[26] N. Trebon and et al. An approximate method for optimizing hpc component applications in the presence of multiple component implementations. Suffix SAND2003-8760C, Sandia National Laboratories, 2003.

[27] C. Ururahy and et al. ALua: Flexibility for parallel programming. *Computer Languages*, 28(2):155–180, 2002.

[28] Z. Yang and et al. An aspect oriented approach to dynamic adaptation. In *the 1st Workshop on Self-healing Systems*, pages 85–92, Charleston, South Carolina, 2002. ACM.