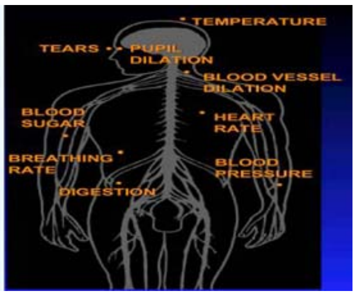
Cluster Computing 9, 5–17, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The United States.

The Autonomic Computing Paradigm

SALIM HARIRI,* BITHIKA KHARGHARIA, HOUPING CHEN, JINGMEI YANG and YELIANG ZHANG
High Performance Distributed Computing Laboratory, University of Arizona

MANISH PARASHAR and HUA LIU
The Applied Software Systems Laboratory, Rutgers, The State University of New Jersey

Autonomic computing is inspired by the human autonomic nervous system that has developed strategies and algorithms to handle complexity and uncertainties, and aims at realizing computing systems and applications capable of managing themselves with minimum human intervention.



1. Introduction

The advances in computing and communication technologies and software tools have resulted in an explosive growth in networked applications and information services that cover all aspects of our life. These services and applications are inherently complex, dynamic and heterogeneous. In a similar way, the underlying information infrastructure, e.g. the Internet, is large, complex, heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. The combination of the two results in application development, configuration and management complexities that break current computing paradigms, which are based on static behaviors, interactions and compositions of components and/or services. As a result, applications, programming environments and information infrastructures are rapidly becoming brittle, unmanageable and insecure. This has led researchers to consider alternative programming paradigms and management techniques that are based on strategies used by biological systems to deal with complexity, dynamism, heterogeneity and uncertainty.

Autonomic computing is inspired by the human autonomic nervous system that handles complexity and uncertainties, and aims at realizing computing systems and applications capable of managing themselves with minimum human intervention. In this paper we first give an overview of the architecture of the autonomic nervous system and use it to motivate our approach to develop the autonomic computing paradigm. We then illustrate how this paradigm can be used to control and manage complex applications.

2. Motivations: The human autonomic nervous system

The human nervous system is, to the best of our knowledge, the most sophisticated example of autonomic behavior existing in nature today [1]. It is the body's master controller that monitors changes inside and outside the body integrates sensory input, and effects appropriate response. In conjunction with the endocrine system, which is the body's second important regulating system, the nervous system is able to constantly regulate and maintain homeostasis. Homeostasis is one of the most remarkable properties of highly complex systems. A homeostatic system (a large organization, an industrial firm, a cell) is an open system that maintains its structure and functions by means of a multiplicity of dynamic equilibriums that are rigorously controlled by interdependent regulation mechanisms. Such a system reacts to every change in the environment, or to every random disturbance, through a series of modifications that are equal in size and opposite in direction to those that created the disturbance. The goal of these modifications is to maintain internal balances.

The manifestation of the phenomenon of homeostasis is widespread in the human system. As an example consider the mechanisms that maintain the concentration of glucose in the blood within limits—if the concentration should fall

* This work is supported in part by grants from NSF/NGS Contract CNS-0305427, NSF/SEI(EAR) Contract 0431079 and Intel Corporation ISTG R&D Council.

below about 0.06 percent, the tissues will be starved of their chief source of energy; if the concentration should rise above about 0.18 percent, other undesirable effects will occur. If the blood-glucose concentration falls below about 0.07 percent, the adrenal glands secrete adrenaline, which causes the liver to turn its stores of glycogen into glucose. This passes into the blood and the blood-glucose concentration drop is opposed. Further, a falling blood-glucose also stimulates the appetite causing food intake, which after digestion provides glucose. On the other hand, if the blood-glucose concentration rises excessively, the secretion of insulin by the pancreas is increased, causing the liver to remove the excess glucose from the blood. Muscles and skin also remove excess glucose and if the blood-glucose concentration exceeds 0.18 percent, the kidneys excrete excess glucose into the urine. Thus, there are five activities that counter harmful fluctuations in blood-glucose concentration [2].

The above example focuses on the maintenance of the blood-glucose concentration within safe or operational limits that have been 'predetermined' for the species. Similar control systems exist for other parameters such as systolic blood pressure, structural integrity of medulla oblongata, severe pressure of heat on the skin and so on. All these parameters have a bearing on the survivability of the organism, which in this case is the human body. However, all parameters are not uniform in their urgency or their relations to lethality. Parameters that are closely linked to survival and closely linked to each other so that marked changes in one leads sooner or later to marked changes in the others, have been termed as essential variables by Ashby in his study of the design for a brain [2], which is discussed below.

2.1. Ashby's ultrastable system

Every real machine embodies no less than an infinite number of variables, and for our discussion we can safely think of the human system as represented by a similar sets of variables, of which we will consider a few. In order for an organism to survive, its essential variables must be kept within viable limits (refer figure 1). Otherwise the organism faces the possibility of disintegration and/or loss of identity (dissolution or death) [3].

The body's internal mechanisms continuously work together to maintain essential variables within their limits. Ashby's definition of adaptive behavior as demonstrated by the human body follows from this observation. He states that a form of behavior is adaptive if it maintains the essential variables within physiological limits [2] that define the viability zone. Two important observations can be made:

1. The goal of the adaptive behavior is directly linked with the survivability of the system.
2. If the external or internal environment pushes the system outside its physiological equilibrium state the system will always work towards coming back to the original equilibrium state.

Ashby observed that many organisms undergo two forms of disturbance: (1) frequent small impulses to main variables and (2) occasional step changes to its parameters. Based on these observations he devised the architecture of the ultra-stable system that consists of two closed loops: one that can control small disturbances while the second control loop is responsible for larger disturbances. This is shown in figure 2.

As shown in figure 2, the ultra-stable system consists of two sub-systems, the environment and the reacting part (R).

R represents a subsystem of the organism that is responsible for overt behavior or perception. It uses the sensor channels as part of its perception capability and motor channels to respond to the changes impacted by the environment. These set of sensor and motor channels constitute the primary feedback between R and the environment. We can think of R as a set of behaviors of the organism that get triggered based on the changes affected by the environment. S represents the set of parameters that trigger changes in relevant features of this behavior set.

Note that in figure 2, S trigger changes only when the environment affects the essential variables in a way that causes them to be outside their physiological limits. As mentioned above, these variables need to be maintained within physiological limits for any adaptive system/organism to survive. Thus we can view this secondary feedback between the environment and R as responsible for triggering the adaptive behavior of the organism. When the changes impacted by the environment on the organism are large enough to throw the essential variables out of their physiological limits, the secondary feedback becomes active and changes the existing behavior sets of the organism to adapt to these new changes. Notice that any changes in the environment tend to push an otherwise stable

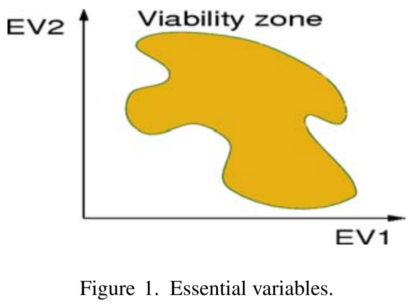


Figure 1. Essential variables.

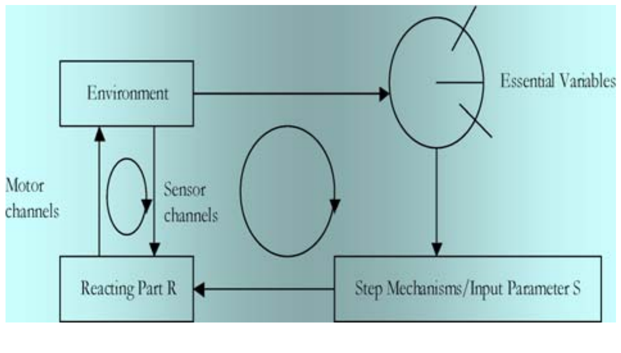


Figure 2. The ultra-stable system architecture [1].

system to an unstable state. The objective of the whole system is to maintain the subsystems (the environment and R) in a state of stable equilibrium. The primary feedback handles finer changes in the environment with the existing behavior sets to bring the whole system to stable equilibrium. The secondary feedback handles coarser and long-term changes in the environment by changing its existing behavior sets and eventually brings back the whole system to stable equilibrium state. Hence, in a nutshell, the environment and the organism always exist in a state of stable equilibrium and any activity of the organism is triggered to maintain this equilibrium.

2.2. *The nervous system: A subsystem within Ashby's ultra-stable system*

The human nervous system is adaptive in nature. In this Section we apply the concepts of Ashby's ultra-stable system to the human nervous system. The goal is to enhance the understanding of an adaptive system and help extract essential concepts that can be applied to the autonomic computing paradigm presented in the following sections.

As shown in figure 3, the nervous system is divided into the Peripheral Nervous System (PNS) and the Central Nervous System (CNS). The PNS consists of sensory neurons running from stimulus receptors that inform the CNS of the stimuli and motor neurons running from the CNS to the muscles and glands, called effectors, which take action. CNS is further divided into two parts: sensory-somatic nervous system and the autonomic nervous system. Figure 4 shows the architecture of the autonomic nervous system modeled after Ashby's ultra-stable system.

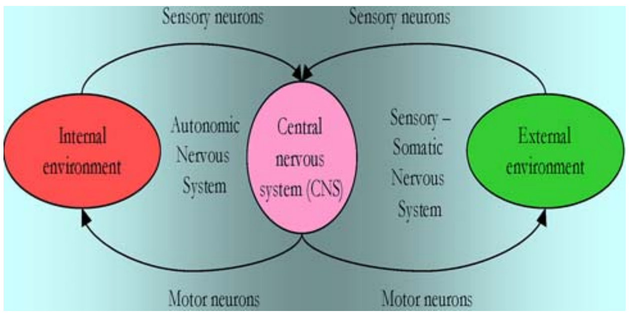


Figure 3. Organization of the nervous system.

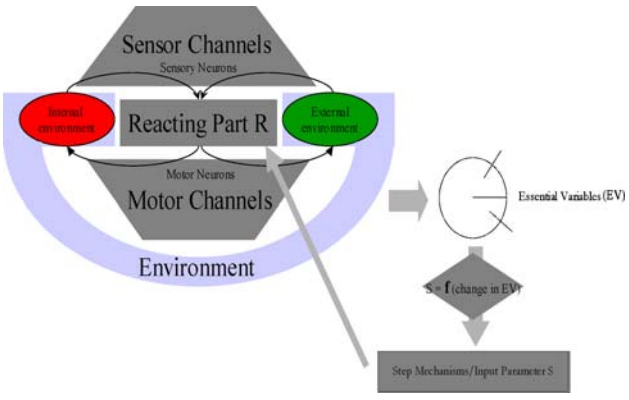


Figure 4. Nervous system as part of an ultrastable system.

As shown in figure 4, the Sensory and Motor neurons constitute the Sensor and Motor channels of the ultra-stable system. The triggering of essential variables, selection of the input parameter S and translation of these parameters to the reacting part R constitute the workings of the Nervous System. Revisiting the management of blood-glucose concentration within physiological limits discussed earlier, the five mechanisms that get triggered when the essential variable (concentration of glucose in blood) goes out of the physiological limits change the normal behavior of the system such that the reacting part R works to bring the essential variable back within limits. It uses its motor channels to effect changes so that the internal environment and the system (organism) come into the state of stable equilibrium. It should be noted that the environment here is divided into the internal environment and external environment. The internal environment represents changes impacted internally within the human system and the external environment represents changes impacted by the external world. However, the goal of the organism is to maintain the equilibrium of the entire system where all the sub-systems (the organism or system itself, the internal and external environments) are in stable equilibrium.

2.3. *The autonomic computing paradigm*

An autonomic computing paradigm, modeled after the autonomic nervous system, must have a mechanism whereby changes in its essential variables (e.g., performance, fault, security, etc.) can trigger changes to the behavior of the computing system such that the system is brought back into equilibrium with respect to the environment. This state of stable equilibrium is a necessary condition for the survivability of the organism. In the case of an autonomic computing system, we can think of survivability as the systems ability to protect itself, recover from faults, reconfigure as required by changes in the environment and always maintain its operations at a near optimal performance. Both the internal (e.g. excessive CPU utilization) and the external environment (e.g. protection from an external attack) impact its equilibrium. The autonomic computing system requires sensor channels to sense the changes in the internal and external environment and motor channels to react to the changes in the environment by changing itself so as to counter the effects of changes in the environment and maintain equilibrium. The changes sensed by the sensor channels have to be *analyzed* to determine if any of the essential variables has gone out of their viability limits. If so, it has to trigger some kind of *planning* to determine what changes to inject into the current behavior of the system such that it returns to equilibrium state within the new environment. This planning would require *wisdom* to select just the right behavior from a large set of possible behaviors to counter the change. Finally, the motor neurons *execute* the selected change. 'Sensing', 'Analyzing', 'Planning', 'Wisdom' and 'Execute' are in fact the keywords used to identify an autonomic system [4]. In what follows, we use these concepts to identify the properties of an autonomic computing system. We then present the architecture of an autonomic computing system being developed at The University of Arizona and Rutgers University, that provides key autonomic middleware services to support the composition and self-managed execution of autonomic applications. We will discuss how simple "un-intelligent" applications are converted into autonomic applications in a process of dynamic and opportunistic composition of autonomic components. We then discuss the execution and management of these applications within our proposed autonomic computing framework.

3. Properties of an autonomic computing system

An autonomic computing system can be a collection of autonomic components, which can manage their internal behaviors and relationships with others in accordance to high-level policies. The principles that govern all such systems have been summarized as eight defining characteristics [5,6]:

1. *Self-Awareness*: an autonomic system knows itself and is aware of its state and its behaviors.
2. *Self-Protecting*: an autonomic system is equally prone to attacks and hence it should be capable of detecting and protecting its resources from both internal and external attack and maintaining overall system security and integrity.
3. *Self-Optimizing*: an autonomic system should be able to detect performance degradation in system behaviors and intelligently perform self-optimization functions.
4. *Self-Healing*: an autonomic system must be aware of potential problems and should have the ability to reconfigure itself to continue to function smoothly.
5. *Self-Configuring*: an autonomic system must have the ability to dynamically adjust its resources based on its state and the state of its execution environment.
6. *Contextually Aware*: an autonomic system must be aware of its execution environment and be able to react to changes in the environment.
7. *Open*: an autonomic system must be portable across multiple hardware and software architectures, and consequently it must be built on standard and open protocols and interfaces.
8. *Anticipatory*: an autonomic system must be able to anticipate, to the extent that it can, its needs and behaviors and those of its context, and be able to manage itself proactively.

Sample autonomic system/applications behaviors include installing software when it detects that the software is missing (self-configuring), restart a failed element (self-healing), adjust current workload when it observes an increase in capacity (self-optimizing) and take resources offline if it detects that these resources are compromised by external attacks (self-protecting).

Each of the attributes listed above are active research areas towards realizing autonomic systems and applications. Generally, autonomic computing addresses these issues in an integrated manner, i.e., configuration, optimizing, protection, and healing. Further, autonomic management solutions typically consists of the steps outlined above: (1) the application and underlying information infrastructure provides information to enable context and self awareness; (2) system/application events trigger analysis, deduction and planning using system knowledge; and (3) plans are executed using the adaptive capabilities of the system. An autonomic system implements self-managing attributes using the control loops described above to collect information, makes decisions and adapt as necessary.

An autonomic application can be viewed as a set of interacting components that component need to collaborate to achieve coherent autonomic behavior. This requires a common set of underlying capabilities including representations and/or mechanisms for solution knowledge, system administration, problem determination, monitoring and analysis, policy definition and enforcement and transaction measurements [7]. For example, a common solution knowledge capability captures install, configuration and maintenance information in a consistent manner, and eliminates the complexity introduced by heterogeneous tools and formats. Common administrative console functions ranging from setup and configuration to runtime monitoring and control provide a single platform to host administrative functions across systems and applications. Hence they enable users to manage solutions rather than managing individual systems/applications. Problem determination is one of the most basic capabilities for autonomic elements and enables it to decide on appropriate actions when healing, optimizing, configuring or protecting itself. Autonomic monitoring is a capability that provides an extensible runtime environment to support the gathering and filtering of data obtained through sensors. Complex analysis methodologies and tools provide the power and flexibility required to perform a range of analyses of sensor data, including deriving information about resource configuration, status, offered workload and throughput. A uniform approach to defining the policies is necessary to support adaptations and govern decision-making required by the autonomic system. Transaction measurements are needed to understand how the resources of heterogeneous systems combine into a distributed transaction execution environment. Using these measurements, analysis and plans can be derived to change resource allocations to optimize performance across these multiple systems as well as determine potential bottlenecks in the system.

4. Autonomic computing system: The conceptual architecture

Figure 5 presents the architecture of an autonomic computing system from the conceptual point of view. This architecture directly derives from Ashby's ultra-stable system (refer figure 2). It consists of the following modules:

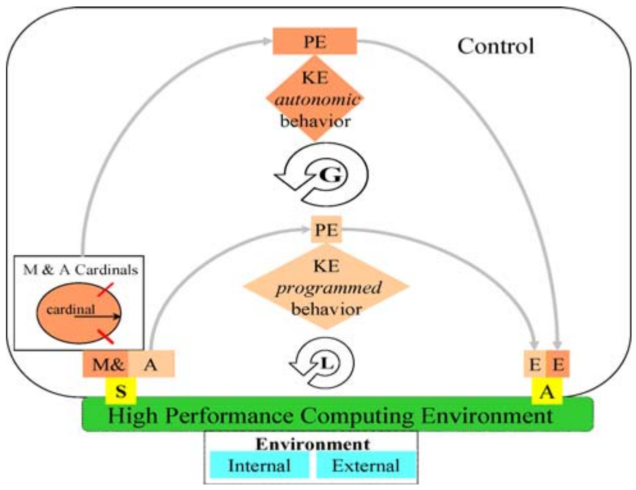


Figure 5. Autonomic computing system: The conceptual view.

4.1. *High performance computing environment*

This comprises of the high performance computing applications and their execution environment. The autonomic system is geared towards self-control and self-management of the high performance computing environment.

4.2. *Environment*

The environment represents all the factors that can impact the high performance computing system. The environment and the high performance computing system can be two subsystems forming a stable system (refer figure 2). Any change in the environment causes the whole system to go from a stable state to an unstable state. This change is then offset by reactive changes in the high performance computing system causing the system to move back from the unstable state to a different stable state. Notice that the environment consists of two parts—internal and external. The internal environment consists of changes internal to the application that characterizes the runtime state of the application. The external environment can be thought of as characterizing the state of the execution environment.

4.3. *Control*

At runtime, the High Performance Computing Environment can be affected in different ways, for example, it can encounter a failure during execution, it can be externally attacked or it may slow down and affect the performance of the entire application. It is the job of the Control to manage such situations as they occur at runtime. The Control has the following engines to execute its functionalities:

1. *Monitoring and Analysis Engine* (*M&A*): Monitors the state of the high performance computing environment through its sensors and analyzes the information.
2. *Planning Engine* (*PE*): Plans alternate execution strategies (by selecting appropriate actions) to optimize the behaviors (e.g. to self-heal, self-optimize, self-protect etc.) and operations of the high performance computing environment.
3. *Knowledge Engine* (*KE*): Provides the decision support to the Control to pick up the appropriate rule from a set of rules to improve the performance.

The Control realizes its control and management objectives with the aid of two closed loop control sub-systems, triggered by application and system sensors (reactive) and online predictive performance models (proactive), and will manage and optimize application execution (refer figure 5).

4.3.1. *Local control loop*

The local or fine control loop will locally manage the behavior of individual and local system elements on which the components execute. This can be viewed as adding self-managing capabilities to conventional components/elements. This loop will control local algorithms, resource allocation strategies, distribution and load balancing strategies, etc. Note that this loop will only handle *known* environment states and the mapping of environment states to behaviors is encapsulated in its *knowledge engine* (*KE*). For example, when the load on the system resources exceeds the acceptable threshold value, the fine loop control will balance the load by either controlling the local resources or by reducing the size of the computational loads. This will work only if the local resources can handle the computational requirements. However, the fine loop control is *blind* to the overall behavior and thus cannot achieve the desired overall objectives. Thus by itself it can lead to sub-optimal behavior.

4.3.2. *Global control loop*

At some point, one of the essential variables of the system eventually exceeds its limits that will trigger the global loop control subsystem. The global control loop will manage the behavior of the overall application and will define the knowledge that will drive the local adaptations. This control loop can handle *unknown* environment states and uses four cardinals for monitoring and analysis of the high performance computing environment i.e., performance, fault-tolerance, configuration and security. These cardinals are analogous to the essential variables described in Ashby's ultra stable system model of the autonomic nervous system [refer figure 2]. This control loop acts towards changing the existing behavior of the high performance computing environment such that it can adapt itself to changes in the environment. For example, in the previous load-balancing scenario, the existing behavior of the high performance computing environment (as directed by the local loop) was to maintain its local load within prescribed limits. Doing so blindly may degrade the performance of the overall system. This change in the overall performance cardinal triggers the global loop. The global loop then selects an alternate behavior pattern from the pool of behavior patterns for this high performance computing environment. The

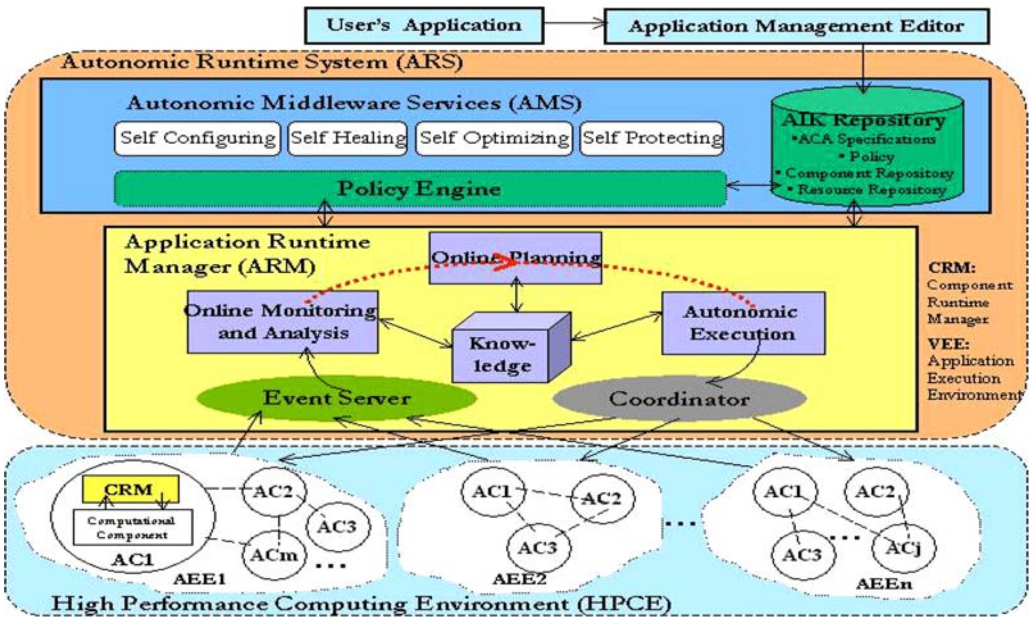


Figure 6. Autonomic computing system: The implementation view.

Planning Engine (PE) determines the appropriate plan of actions using its Knowledge Engine (KE). Finally the Execution Engine (EE) executes the new plan on the high performance computing environment in order to adapt its behavior to the new environment conditions.

The most important feature of the autonomic system is the integrated approach of its controller. The controller unit manages in an integrated manner performance, fault, security and configuration of computing systems and their applications. In the classical paradigm, each of these properties has been treated separately and in isolation. These practices have contributed significantly to the control and management challenges of large scale interacting and dynamic computing systems and services. In the next section, we describe our approach to implement an autonomic computing system based on the conceptual architecture.

5. Autonomic computing system frame work

Figure 6 illustrates the main modules to implement our autonomic computing system framework that is inspired by the Ashby's ultra-stable system architecture (refer Section 2.1).

5.1. Application management editor

In this framework, the users and/or application developers can specify the characteristics and requirements of their applications using an Application Management Editor (AME). Each application can be expressed in terms of one or more autonomic components. These components are expressed using the Autonomic Component as will be discussed later. Once the application is specified, the next step is to control and manage the execution of the application at runtime using the services and routines offered by the Autonomic Runtime System (ARS).

5.2. Autonomic Runtime System (ARS)

The Autonomic Runtime System (ARS) exploits the temporal and heterogeneous characteristics of the scientific computing applications, and the architectural characteristics of the computing and storage resources available at runtime to achieve high-performance, scalable, and robust scientific simulations. ARS will provide appropriate control and management services to deploy and configure the required software and hardware resources to autonomously (e.g., self-optimize, self-heal) run large-scale scientific applications in computing environments. The local control loop is responsible for control and management of one autonomic component, while the global control loop is responsible for the control and management of an entire autonomic application.

The ARS can be viewed as an application-based operating system that provides applications with all the services and tools required to achieve the desired autonomic behaviors (self-configuring, self-healing, self-optimizing, and self-protection). The primary modules of ARS are the following:

5.2.1. Application Information and Knowledge (AIK) repository

The AIK repository stores the runtime application status information, the application requirements, and knowledge about optimal management strategies for both applications and system resources that have proven to be successful and effective. In addition, AIK contains *Component Repository* (*CR*) that stores the components that are currently available for the users to compose their applications with and Resource Repository

```
1 While (Component $ACA_i$ is running) do
2     State = $CRM_i$ Monitoring ($ACA_i$)
3     State_Deviation = State_Compare(State, DESIRED_STATE)
4     If (state_deviation == TRUE)
5         CRMi Send_Event(State)
6         Event Server Notify ARM
7         Event_Type = CRM_Analysis (State)
8         If ($CRM_{i\ ABLE}$) Then
8             Actions = CRM_Planning(State, Event_Type)
9             Autonomic_Service $AS_j \in \{AS_{config}, AS_{heal}, AS_{optimization}, AS_{security}\}$
10            Execute $AS_j$ (Actions)
11        Else
12            Actions = ARM_Analysis (State, Event_Type)
13            Execute $As_j$ (Actions)
14        EndIf
15    EndIf
16 EndWhile
```

Figure 7. Self-management algorithm.

(RR) that keeps track of all resources that are currently registered in the environment.

5.2.2. *Autonomic Middleware Services (AMS)*

The AMS provides appropriate control and management services to deploy and configure the required software and hardware resources to run autonomously (e.g., self-optimize, self-heal etc.). These runtime services maintain the autonomic properties of applications and system resources at runtime. To simplify the control and management tasks, we dedicate one runtime service for each desired attribute or functionality such as self-healing, self-optimization, self-protection, etc. The Event Server notifies the appropriate runtime service whenever its events become true. The algorithm that is used to achieve self- management for any service (self-healing, self-optimizing, self-protecting, etc.) is shown in figure 7.

The Component Runtime Manager (CRM) monitors the state of each active component to determine if there are any severe deviations from the desired state (steps 1–3). When an unacceptable change occurs in the component behavior, CRM [12] generates an event into the Event Server, which notifies ARM (steps 4–6). Furthermore, CRM analyzes the event and determines the appropriate plan to handle that event (steps 7 and 8) and then executes the appropriate self-management routines (steps 9–10). However, if the problem cannot be handled by the CRM, the ARM is invoked to take the appropriate management functions (steps 12–13) at a higher granularity (e.g., migrate the components to another machine due to failure or degradation in performance).

5.2.3. *Application Runtime Manager (ARM)—The global control loop*

The ARM focuses on setting up the application execution environment and then maintaining its requirements at runtime. The ARM performs online *monitoring* to collect the status and state information using the component sensors. It *analyzes* component behaviors and detects any anomalies or state changes reflected by out-of-bound values of the cardinals (for example, degradation in performance, component failure). The ARM generates the appropriate control and management *plans* as specified by the *knowledge* in its knowledge repository. The plan generation process involves triggering the appropriate rule from the list of rules (for the appropriate service) in the knowledge repository. The ARM main modules include (1) Online Monitoring and Analysis, (2) Online Planning, and (3) Autonomic Execution.

5.2.4. *Event server*

The Event server receives events from the Component Managers (described in Section 5.3.2) that monitor components and systems and then notifies the corresponding engines that subscribed to these events.

5.2.5. *Coordinator*

The Coordinator is responsible for handling application execution by coordinating between different AC (refer Section 5.3.1) across different coupled models.

5.3. *High Performance Computing Environment (HPCE)*

The HPCE consists of the high performance computing application and the execution environment on which it runs. The HPCE includes the following modules:

5.3.1. *Autonomic components:*

An autonomic component (AC) is the fundamental building block for autonomic applications in our Autonomic Computing Framework (refer figure 6). An autonomic component is a simple computational component with encapsulated rules, constraints and mechanisms for self-management and dynamic interactions with other components. It extends traditional components [8,9] to define a self-contained modular software unit of composition with specified interfaces and explicit context dependencies. The architecture of an autonomic component is shown in figure 8 [10].

The AC implements three different interfaces for importing/exporting the functionalities of the computational component, sensing and changing the runtime state of the component

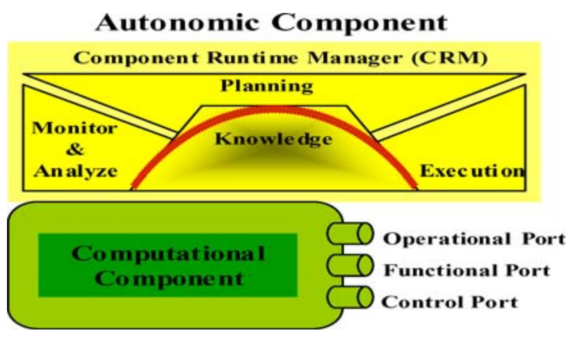


Figure 8. Autonomic computing architecture.

whenever is required and for managing all the attributes (e.g., performance, fault, security) of the computational component.

Functional port (Γ)*:* defines a set of functionalities provided and used by the autonomic component. $\gamma \in \Omega \times \Lambda$, where Ω is the set of inputs and Λ is the set of outputs of the components, and γ defines a valid input-output set.

Control port (Σ)*:* is the set of tuples (σ, ξ), where σ is a set of sensors and actuators exported by the component, and ξ is the constraint set that controls access to the sensors/actuators. Sensors are interfaces that provide information about the component while actuators are interfaces for modifying the state of the component. Constraints are based on state, context and/or high-level access polices, and can control who invokes the interface, when and how they are invoked.

Operational port (Θ)*:* defines the interfaces to formulate, inject and manage rules, and encapsulates a set of rules that are used to manage the runtime behavior of the autonomic component. Rules incorporate high-level guidance and practical human knowledge in the form of conditional if-then expressions, i.e., *IF condition THEN actions*. *Condition* is a logical combination of component (and environment) sensors and events. *Actions* consist of a sequence of invocations of components and/or system sensors/actuators, and other interfaces. A rule fires when its condition expression evaluates to be true which causes the corresponding actions to be executed. Two types of rules are defined here.

*Behavior rules***:** control the runtime functional behaviors of autonomic components and applications. For example, behavior rules can control the algorithms, data representations or input/output formats used by a component and an application.

*Interaction rules***:** control the interactions between components, between components and their environments, and the coordination within an autonomic application. For example, an interaction rule may define where a component will get inputs and forward outputs, define the communication mechanisms used, and specify when the component interacts with other components.

5.3.2. Component Runtime Manager (CRM)—The local control loop

Each autonomic component has its own manager that is delegated to manage its execution. It is the local control subsystem described above. Its Monitoring and Analysis Engine monitors the component and its context by using the Control (sensor) port interface and analyzes that information to determine the state of the component. The Planning Engine then checks to see if that *state* (the *condition* in the '*IF condition THEN action*) matches with any of the states for this component as stored in the Knowledge Engine. If a match is detected, the PE picks up the *action* corresponding to that *state* (condition) and the Execution Engine executes that action on the autonomic component using its Control (actuator) port interface. In other words one rule from all the set of rules for this component is fired. These rules are stored in the Knowledge Engine. Note that each rule is already assigned a priority level. The Planning Engine uses this information to manage multiple firings and resolve conflicts [11]. As can be seen here, the local control loop is only capable of handling known application and environment states.

5.3.3. Dynamic composition of autonomic components: Coupled Component Architecture (CCA)

As described above, the application is developed as a composition of smaller autonomic components. This fact is depicted in figure 6 where multiple ACs are coupled together to form one coupled model. At runtime, as the execution proceeds, new components might be added to the application workflow and/or old components might be deleted. Thus the application workflow dynamically changes its structure at runtime. The autonomic components should have the capability of coupling and interacting with other autonomic components just as newly born cells within the human body, adapt into the existing cell-system automatically and dead cells are *gracefully* deleted.

The composition of autonomic components consists of defining *an organization of components* and the *interactions among these components*. The organization of components is based on the composition of functional ports (Γ), and can be defined as:

$$\mathrm{C}_0 \propto\ _\Gamma \cup \mathrm{Ci}, \quad \exists \Gamma_{C0,u} \subseteq \cup \Gamma_{Ci,p}$$

where C_0 is an autonomic component, $\cup$ Ci is a set of one or more autonomic components, $\propto_\Gamma$ denotes the relation "be functionally composable with", $\Gamma_{C0,u}$ is the functions used by component C_0, and $\cup\Gamma_{Ci,p}$ represents the functions provided by the component set $\cup$ Ci. This definition says that component C_0 is functionally composable with components $\cup$ Ci, when $\cup$ Ci can provide all the functions required by C_0. This is similar to the composition defined by component-based frameworks such as the CORBA [9] and Web services.

Interactions among components define how and when components interact, the interaction mechanism (messaging, shared-memory, tuple-space) and coordination model (data-driven or control-driven). For example, Ccaffeine [12,13] defines interactions as function calls, CORBA [9] uses remote method invocations, and Web services and Grid services [14] communicate using XML messages. Interactions may be triggered by an event or may be actively initiated by a component.

Dynamic composition introduces dynamism and uncertainty into both aspects of composition described above, i.e., "which components are composed" and "how and when they interact" are defined only at runtime. Compositions are often represented as workflow graphs where nodes represent components and edges represent interaction relationships between the components. Using such a workflow graph representation of composition, dynamic composition consists of (a) node (component) dynamism—components are replaced, added to or deleted from the workflow, and (b) edge (interaction) dynamism—interaction relationships are changed, added to or deleted from the workflow.

In a HPCE there could be several autonomic applications running. Each autonomic application has its own computing environment that we refer to it as Application Execution Environment (AEE) and managed by the ARM as shown in figure 6.

6. Illustrative example: Distributed wildfire simulation

In this section we introduce a wildfire simulation and then show how we can automate such an application using our Autonomic Computing Architecture and the proposed framework.

6.1. Distributed wildfire simulation

The wildfire spread model, among other things predicts average fire spread as the fire propagates based on both static and dynamic conditions and allows uncertainty to be incorporated into the model by having certain fire spread input parameters to be modeled as random variables sampled from arbitrary probability distributions. This allows for a more realistic approach to setting fire spread parameters that cannot be determined with certainty. The model also allows for simple rules for forest fire fighting scenarios to be implemented and thus provides a flexible platform for studying the effect of pouring water on forest cells.

In our forest fire model, the forest is represented as a 2-D cell-space composed of cells of dimensions $l \times b$ (l: length, b: breadth). For each cell there are eight major wind directions N, NE, NW, S, SE, SW, E, W.In our architecture, a group of such individual cells will together constitute, a Computational Unit (CU). The weather and vegetation conditions are assumed to be uniform within a cell, but may vary in the entire cell space. A cell interacts with its neighbors along all the eight directions as listed in figure 9.

When a cell is ignited, its state will change from "unburned" to "burning". During its "burning" phase, the fire will propagate to its eight neighbors along the eight directions. The direction and the value of the maximum fire spread rate within the burning cell can be computed using Rothermel's fire spread model [15], which takes into account the wind speed and direction, the vegetation type, the fuel moisture and terrain type, such as slope and aspect, in calculating the fire spread rate. When the simulation time advances to the ignition times of neighbors, the neighbor cells will ignite and their states will change from "unburned" to "burning". In a

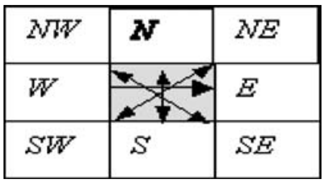


Figure 9. Fire directions after ignition.

similar way, the fire would propagate to the neighbors of these cells.

The wildfire simulation model used in this paper is based on fireLib [16], which is a C function library for predicting the spread rate and intensity of free-burning wildfires. We parallelized the sequential version of the fire simulation using MPI. This parallelized fire simulation divides the entire cell space among multiple processors such that each processor works on its own portion and exchanges the necessary data with each other after each simulation time step.

6.2. Autonomic wildfire simulation

In this Section we will describe how to apply our Autonomic Computing Framework to autonomously run the wildfire simulation as shown in figure 10.

6.2.1. General process

The Autonomic Runtime Manager is responsible for setting up the execution environment for the wildfire application. Once the application is running, ARM will manage the application execution at runtime to improve performance, accuracy and scalability.

The ARM main modules include (see figure 10) 1. Online Monitoring and Analysis, 2. Autonomic Planning and Scheduling. The Online Monitoring module (step 1,2 and 6) interfaces with different sensors located on each resource involved in the execution of the wildfire simulation. Those sensors monitor the current state of the fire simulation in terms of the number and the locations of burning cells and unburned cells. In addition, the sensors monitor the states of the resources, such as the CPU load, available memory, network load etc. The runtime state information is stored in a database. The Online Analysis module (step 3) analyzes the runtime usage information of the wildfire simulation and then determines whether or not the current workload distribution needs to be changed.

The Autonomic Planning engine partitions the wildfire simulation domain into sub-domains, such as Natural Regions (*NRs*), where each region has the same temporal and spatial characteristics (e.g., burned (*NR1*), burning (*NR2*), and unburned regions (*NR3*)). Based on the objectives of the analysis (e.g., accurate vs coarse simulations), the Autonomic Planning and Scheduling engine will use the resource capability models as well as performance models associated with the computations, and the knowledge repository to select the appropriate wildfire models and parameters for each region and then decompose the computational workloads for each Natural Region into schedulable Computational Units (*CUs*) (step 4). Based on the availability of computing resources and their access policies, the Scheduler will dynamically schedule the *CUs* to run on clusters of high performance workstations, massive parallel computers, and/or distributed/shared memory multiprocessor systems (step 5). It is to be noted here that the ARM hides the underlying heterogeneity of the execution environment from the user and can interface the wildfire

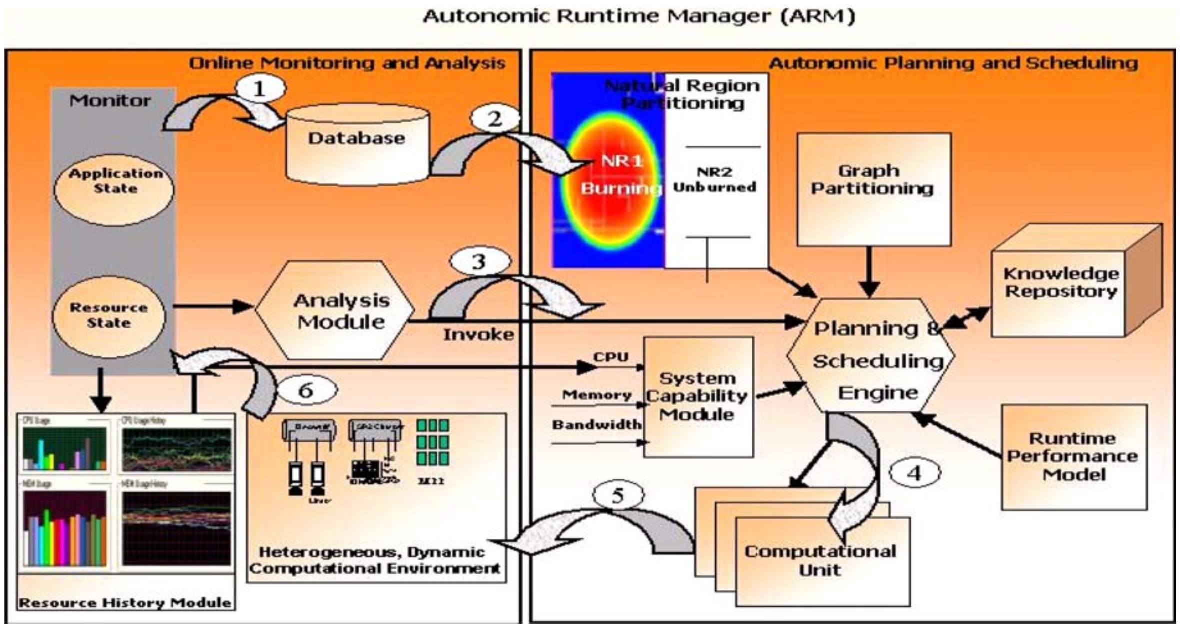

Autonomic Runtime Manager (ARM)
Online Monitoring and Analysis
Autonomic Planning and Scheduling
Monitor
Application State
Resource State
Database
Analysis Module
Invoke
Natural Region Partitioning
NR1 Burning
NR2 Unburned
Graph Partitioning
Knowledge Repository
Planning & Scheduling Engine
CPU
Memory
Bandwidth
System Capability Module
Runtime Performance Model
Computational Unit
Heterogeneous, Dynamic Computational Environment
Resource History Module
1
2
3
4
5
6

Figure 10. Wildfire spread execution within the autonomic computing framework.

simulation to different types of execution models and different types of resources harnessing the maximum utilization of features and capabilities of the underlying environment.

6.2.2. Specific scenario

In this Section, we will apply the process described in the previous section to the specific scenario of wildfire spread shown in figure 11 and explain it in the light of figure 10. Figure 11 shows the state of the wildfire simulation application at three different instants of simulation time.

The user first composes the application and submits its specifications using the framework application management editor. In this step, the user defines the main tasks/components and the specifications of each component such as—what vegetation map to use, which terrain map to use, which wind models to use, what is the problem size, what is the initial ignition point, number of processors, OS and memory requirements and so on. This information is stored in the AIK Repository (refer Section 5.2.1). In addition, users define the policies that must be followed to control and manage the application at runtime. To do the planning efficiently, the ARM uses the knowledge repository that stores the rules that need to be followed for each condition and identifies the rules that must be fired and consequently the actions that must be taken.

In this application, the entire cell space is divided into smaller groups of cells and distributed on available resources for execution. Each such group of cells form one or more Computational Units (CU) that are defined according to the Autonomic Component Architecture (refer Section 5.3.1).

With reference to figure 10, the Autonomic Runtime Manager consists of two main modules. We explain below, the roles played by each module in the execution and runtime management of the wildfire simulation as shown by the three scenarios in figure 11.

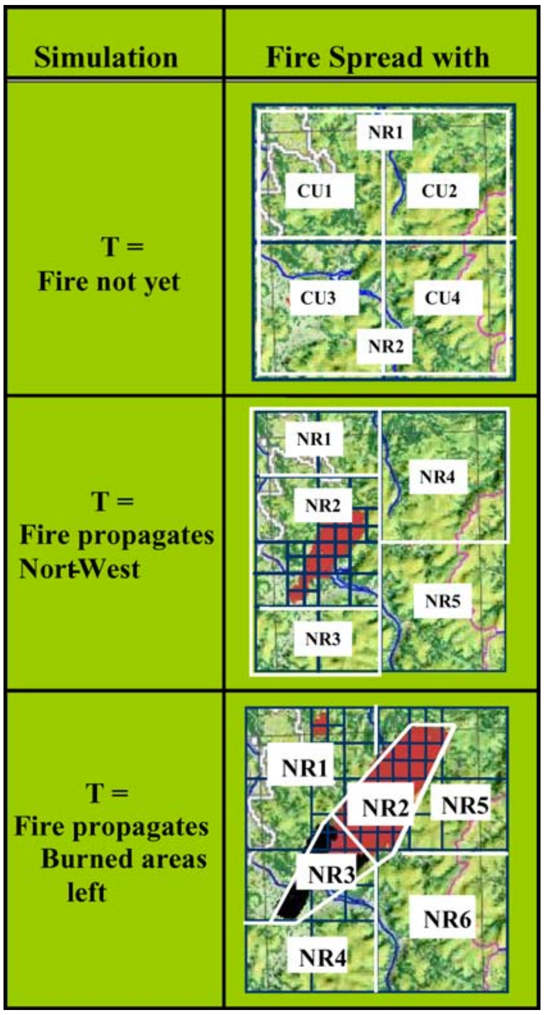


Figure 11. Fire propagation with time. The grid reflects resource allocation plan to the application at runtime.

(a) *Online Monitoring and Analysis.* The Online Monitoring engine (refer figure 10) senses the state of the application and the resources by using its sensors. The Analysis engine uses this monitored information to determine whether the repartitioning is needed. This module performs steps 1, 2, 3 and 6 as shown in figure 10.

Step 1, 2 and 6: *Monitoring Module*: The Online Monitoring engine senses the state of the application and the resources at runtime. The framework uses that information to execute it's runtime management functions.

Step 3: *Analysis Module*: The online analysis module analyzes the monitored information and the resource load information of the wildfire simulation pieces and then determines whether or not re-partitioning is needed. The rule to be triggered in this case is

IF Condition: *load imbalance* > *threshhold*
Then Action: *Repartion*

(b) *Autonomic Planning and Scheduling.* The ARM self-optimizes the application execution by using the online Planning engine that takes into account the application state as well as the state of underlying computing resources.

Step 4: *Partitioning*: The autonomic planning module partitions the cell domain into sub-domains based on the current states of the application as well as the predicted states of the next simulation steps. Different partitioning methods can be integrated with the planning engine. The Natural Region method partitions the cell space into Natural Regions (*NRs*)(e.g., burned (*NR1*), burning (*NR2*), and unburned regions (*NR3*))**.** The Graph partitioning method [17] represents the cell space as an undirected weighted graph with cells as vertices and the computational complexity as weights. Thus a graph partitioning for the fire simulation domain yields the assignment of cells to processors. The planning engine also uses a predictive model to estimate the cells that will be burning in the next simulation steps. Below we use the Natural Region method to illustrate the partitioning process.

The initial partitioning is performed once before the start of the execution. It uses the initial application information provided by the user and the initial state of the execution environment. For example, in figure 11 at time $T = 0$, the fire has not yet started. Hence the partitioning into NR is done based on the vegetation type of the cells and the terrain slope. In figure 11, the cell space is partitioned into 2 NR. Each NR has cells with the same or nearly the same vegetation type and terrain topography. If NR1 has a steeper slope and vegetation that burns faster, at runtime its resource requirement will be more than that of NR2.

Repartitioning is performed more than once, dynamically at runtime. It uses the runtime state of the CUs and that of the resources to do the repartitioning as required. For example, in figure 11 at time $T = N$, the fire starts propagating such that we have some burning cells (in red) and some unburned cells (in green). The cell space is partitioned into five NRs such that the burning area falls under one NR. Since the burning cells perform maximum computation, NR2 (refer figure 11) will require maximum number of resources at runtime. Consequently, the Planning engine uses this information to change its resource allocation plan such that NR2 gets the maximum resources

Based on the objectives of the analysis (e.g., accuracy vs speed), the Planning engine uses the resource capability models as well as runtime performance models associated with the computations, and the knowledge repository to select the appropriate components and parameters for each region and then decompose the computational workloads for each natural region into schedulable Computational Units (*CUs*). Since CUs are the actual schedulable computational units, the Planning engine partitions NR2 into finer grids (where each box represents a CU—refer figure 11). This implies that NR2 has the largest number of CUs as compared to the rest of the cell-space.

Similarly, in figure 11 at time $T = 2N$, some of the cell-space has burned while some are burning and the rest unburned. Since the burned cells do not perform as much computation as they used to when they were burning, hence the idle compute cycles wasted by them can be put to better use by allocating them where required. Thus, as shown in figure 11, NR3 is divided into four CUs as compared to twenty-seven (27) CUs in NR2.

Step 5: *Autonomic Scheduling.* Based on the availability of computing resources and their access policies, the scheduler will dynamically schedule the CUs on clusters of computing resources.

6.2.3. Experimental results and evaluation

Our experiments show that our autonomic computing framework can significantly improve the performance of the wild fire simulation. We have evaluated two application sizes. The first problem size is a $256 * 256$ cell space with 65536 cells (64 K). The second problem has a $512 * 512$ cell domain with 262144 cells (256 K). To introduce heterogeneous fire patterns, the fire is started in the southwest region of the domain and then propagates northeast along the wind direction until it reaches the edge of the domain. In order to make the evaluation for different problem sizes accurate, we maintain the same ratio of burning cells to 17%; that is the total number of burning cells when the simulation terminates is about 17% of the total cells for both problem sizes. The experiments were conducted with different number of processors on a Beowulf cluster.

Figure 12 shows the overall execution time with different runtime partitioning approaches as well as the static scheduling algorithm for two problem sizes and for different processor configurations. For problem size 65536 cells on 8 processors (see figure 12(a)) the ARM system with graph partitioning approach provides an improvement of 45% over static scheduling, 3% over runtime optimization with the natural region

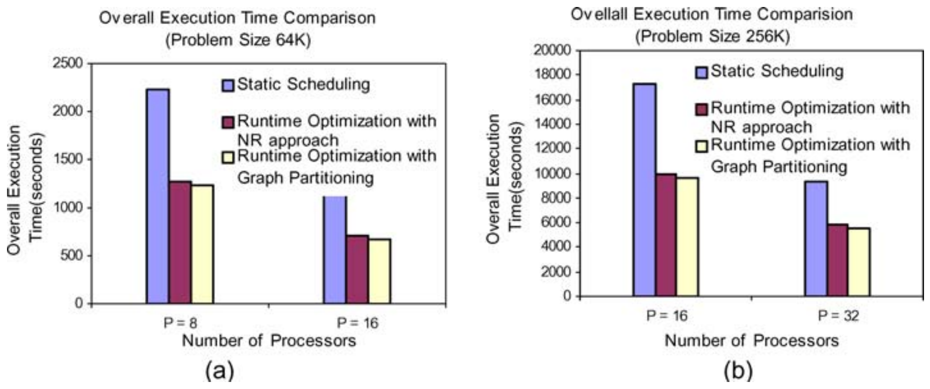


Figure 12. Overall performance of different optimization approaches on different processor configurations. (a) Problem size 65536 cells. (b) Problem size 262144 cells.

partitioning approach. On 16 processors, these numbers are 45% and 6%. For problem size 262144 cells on 16 processors (see figure 12(b)), the numbers are 44% and 3%, while, on 32 processors, they are 41% and 5%.

7. Research issues

Autonomic computing draws most of its principles from the human autonomic nervous system that has developed successful strategies to handle unprecedented complexity, heterogeneity and uncertainty. We strongly believe autonomic computing will lead to breakthrough in the way we design and develop computing systems and applications/services. Computing research has evolved in many isolated, loosely coupled research fields as (security, fault tolerance, high performance, AI, network, agent systems, etc.). Each discipline has managed to deliver computing systems and services that meet their target domains (e.g, High Performance Computing, Fault Tolerance Computing). However, if the computing systems and application would require to combine these capabilities; i.e.,deliver computing systems that provide high performance, secure, fault-tolerance, intelligent computing systems and applications are not practically feasible and available. Autonomic computing is the emerging computing field that addresses all these issues in an integral way and can be viewed as the computing field that will converge all these disciplines in one filed – Autonomic Computing. However, there are still many challenges that must be addressed by the research community. In what follows, we highlight some of these challenges (these are by no means a complete list).

Programming paradigm research challenges

- Autonomic components and applications—how do we express the autonomic properties such as self-healing, self-optimizing, self-protection, etc. and how to integrate that with new or existing components and applications.
- Adaptive Compositions of Applications—how can we dynamically add, delete, and change the algorithm used to implement each component at runtime
- Autonomizing Existing Applications/Tasks—how do you convert an existing application and/or a task to become autonomic.

Middleware/runtime research challenges

- Monitoring Services—how do we obtain information about system and application states using off-the-shelf Operating Systems.
- Check pointing of Systems and Applications.
- Integration Challenges—How do we integrate AI adaptive learning, data mining, performance modeling, and apply them to self-control and management of applications and system resources at runtime.
- Self-Configuring and Self-Deployment—how do we dynamically configure/deploy heterogeneous computing network and storage resources at runtime?
- Improving network-monitoring functions to protect, detect potential threats, and achieve a level of decision-making that allows for the redirection of key activities or data.

References

[1] Autonomic Nervous System, http://users.rcn.com/jkimball.ma.ultranet /BiologyPages/P/PNS.html#autonomic
[2] W. Ross Ashby, Design for a brain (Second Edition Revised 1960), published by Chapman & Hall Ltd, London.
[3] AdaptiveSystems, http://www.cogs.susx.ac.uk/users/ezequiel/AS/ lectures
[4] J.O. Kephart and D.M. Chess, The vision of autonomic computing, IEEE Computer 36(1) (2003) 41–503.
[5] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/, October 2001.
[6] IBM Corporation. Autonomic computing concepts, http://www-3.ibm.com/autonomic/library.shtml , 2001.
[7] IBM An architectural blueprint for autonomic computing, April 2003.
[8] Common Component Architecture Tutorial, http://acts.nersc.gov/ events/Workshop2003/slides/CCA/.
[9] C. Szyperski, D. Gruntz, et al. *Component Software Beyond Object-Oriented Programming*, (Great Britain, Addison-Wesley, 2002).
[10] H. Liu, M. Parashar and S. Hariri, A component-based programming framework for autonomic applications, in: *Proceedings of the 1st IEEE*

International Conference on Autonomic Computing (ICAC-04), (IEEE Computer Society Press, New York, NY, USA, May 2004).

[11] H. Liu and M. Parashar DIOS++: A Framework for rule-based autonomic management of distributed scientific applications, in: *Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Lecture Notes in Computer Science*, (Klagenfurt, Austria, Springer-Verlag, 2003).

[12] M. Agarwal, V. Bhat, et al., AutoMate: Enabling autonomic applications on the grid. in: *Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop*, (Seattle, WA, 2003)

[13] B.A. Allan, R. C. Armstrong, et al., The CCA core specifications in a distributed memory SPMD framework. Concurrency: Practice and Experience 14(5) (2003) 323–345.

[14] I. Foster, C. Kesselman, et al., The physiology of the grid: An open grid services architecture for distributed systems integration, Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[15] R.J. Rothermel, A mathematical model for predicting fire spread in wildland fuels, Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station, 1972.

[16] http://www.fire.org

[17] G. Karypis and V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering 48(1) (1998) 71–95.

Salim Hariri received the MSc degree from Ohio State University in 1982 and the Ph.D. degree in computer engineering from the University of Southern California in 1986. He is a professor in the Electrical and Computer Engineering Department at the University of Arizona and the director of the Center for Advanced TeleSysMatics(CAT)): Next-Generation Network-Centric Systems. He is the editor in chief for *Cluster Computing: The Journal of Networks, Software Tools, aand Applications.* His current research focuses on high performence distributed computing, agent-based proactive and intelligentt network menagment systems, design and analysis of high speed networks, and developing software design tools for high performnce computing and communication systems and applications. He has coauthored more than 200 journal and conference research papers and co-author/editor of three books on parallel and distributed computing. He is a member of IEEE Computer Society.
E-mail: hariri@ece.arizona.edu

Bithika Khargharia is a Ph.D. candidate at the High-Performance Distributed Computing lab at the University of Arizona, Tucson, Arizona. Her research focus area is Autonomic Computing, Grid Computing and Programming Paradigms. Currently, she is at Intel's System Technology Lab, investigating tools and techniques related to self-configuration oriented computing systems to support dynamic provisioning and scale-out virtualization. Her research interests include Grid Computing, Computational Intelligence, AI, Scientific Visualization and Discrete Event Systems (DEVS) Formalism. Bithika graduated with a Master's degree in Computer Engineering in 2003 from University of Arizona, Tucson, Arizona and B.S. degree in Electrical Engineering in 2000 from Assam Engineering College.
E-mail: bithika_k@ece.arizona.edu

Huoping Chen received his B.S. degree from North China University of Technology, China in 1994 and M.S. degree from Beijing Polytechnic University, China in 1997. From 1998 to 2002, he worked at Bell Labs China, Lucent Technology as software architect. He is currently a Ph.D. student at the HPDC Lab, ECE department of the University of Arizona. His research interests are Autonomic Computing and Grid Computing, especially on self-configuration and self-adaptation.
E-mail: hpchen@ece.arizona.edu

Jingmei Yang received her B.S. and M.S. degrees in engineering from Beijing Institute of Technology, China in 1995 and 1998, respectively. From 1998 to 2001, she worked as a software engineer at Institute of Software, Chinese Academy of Sciences. She is currently a Ph.D. student at Electrical and Computing Engineering department of the University of Arizona. Her research interests focused on high performance distributed computing, grid computing, autonomic computing and autonomic distributed scientific applications. She is a student member of the IEEE.
E-mail: jm_yang@ece.arizona.edu

Yeliang Zhang received his B. S. degree in Shanghai University, Shanghai, P. R. China in 1996, and his M. S. degree from The University of Arizona in 2000, both in computer science. He is now a Ph.D. candidate in Electrical and Computer Engineering Dept. at The University of Arizona. His research interests include performance evaluation, load balancing and optimization, parallel algorithm, distributed system and autonomic computing. He is a student member of the IEEE Computer Society.
E-mail: zhang@ece.arizona.edu

Manish Parashar is Professor of Electrical and Computer Engineering at Rutgers University, where he also is director of the Applied Software Systems Laboratory. He received a B.E. degree in Electronics and Telecommunications from Bombay University, India and M.S. and Ph.D. degrees in Computer Engineering from Syracuse University. He has received the Rutgers Board of Trustees Award for Excellence in Research (2004–2005), NSF CAREER Award (1999) and the Enrico Fermi Scholarship from Argonne National Laboratory (1996). His research interests include autonomic computing, parallel & distributed computing (including peer-to-peer and Grid computing), scientific computing, software engineering. He is also a member of the IEEE Computer Society Distinguished Visitor Program (2004–2007).
E-mail: parashar@caip.rutgers.edu

Hua Liu received her Ph.D. degree in Computer Engineering from Rutgers University in 2005, M.E. and B.S. from Beijing University of Posts & Telecoms in 2001 and 1998. Her research interests include Parallel and Distributed Computing (Autonomic, Grid, P2P Computing), High-performance Computing, Component and Service Oriented Software.
E-mail: marialiu@caip.rutgers.edu