Coordinating Energy-aware Administration Loops Using Discrete Control

Soguy Mak-Karé Gueye
LIG / UJF
Grenoble, France
soguy-mak-kare.gueye@inria.fr

Noël De Palma
LIG / UJF
Grenoble, France
noel.de_palma@inria.fr

Eric Rutten
LIG / INRIA
Grenoble, France
eric.rutten@inria.fr

***Abstract*—The increasing complexity of computer systems has led to the automation of administration functions, in the form of autonomic managers. One important aspect requiring such management is the issue of energy consumption of computing systems, in the perspective of green computing. As these managers address each a specific aspect, there is a need for using several managers to cover all the domains of administration. However, coordinating them is necessary for proper and effective global administration. Such coordination is a problem of synchronization and logical control of administration operations that can be applied by autonomous managers on the managed system at a given time in response to events observed on the state of this system. We therefore propose to investigate the use of reactive models with events and states, and discrete control techniques to solve this problem. In this paper, we illustrate this approach by integrating a controller obtained by synchronous programming, based on Discrete Controller Synthesis, in an autonomic system administration infrastructure. The role of this controller is to orchestrate the execution of reconfiguration operations of all administration policies to satisfy properties of logical consistency. We apply this approach to coordinate energy-aware managers for self-optimization and self-regulation of processor frequency.**

***Keywords*-autonomic computing, coordination of multiple autonomic managers, modeling, synchronous programming, discrete controller synthesis.**

I. INTRODUCTION

A. Green computing and the need for administration loops

The increasing complexity of computer systems, integrating several distributed components operating in a heterogeneous and dynamic environment, had led to a problem of hand administration to be time-consuming, expensive, and error-prone. In response to this problem, many research works contribute to the automation of administration functions, in the form of autonomic managers.

One important aspect requiring such management is the issue of energy consumption of computing systems, in the perspective of green computing. Its dynamic management is based on the fact that the deployment and configuration of systems can modified in response to changes in workload, infrastructure and resource availability, or power supply. A variety of mechanisms can be designed for power-aware administration, using the autonomic loop framework. For example, they can contribute at the level of processor frequency, or at the level of server provisioning.

When multiple loops run concurrently, their interactions have to be managed themselves, in order to avoid side-effects annihilating the management actions. Our work focuses on this problem, and proposes a solution for the coordination and synchronization of administration managers, seen themselves as manageable elements.

B. Autonomic administration loops

Autonomic computing [9] aims at providing self-management capabilities to systems. As shown in Figure 1, the managed system or resource is monitored through sensors, and an analysis of this information is used, in combination with knowledge about the system, to plan and execute reconfigurations, through the administration actions offered by the system API.

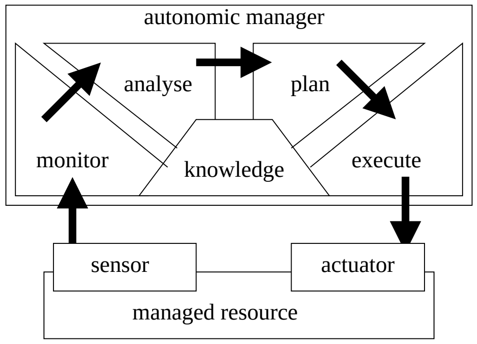


Figure 1. Architecture of an autonomic system

Typical self-management issues handled in this framework are self-configuration, self-optimization, self-healing (fault tolerance and repair), and self-protection. They are managed in closed loop, for which one design methodology is to apply techniques from control theory, continuous or discrete.

C. The problem of coordinating administration loops

Classically, an autonomic manager focuses on one specific concern of system administration. Often, several autonomic managers must be used concurrently to cover all the administration domain. However, using multiple autonomic managers is not enough for ensuring a correct and efficient global system administration. The administration policy followed by each autonomic manager does not take into account the objectives of others aspects: this can of course lead to inconsistencies. In order to benefit from the re-use of several existing autonomic managers, one has to care for coordinating their executions, according to global criteria and properties of their assembly. Most of the proposed solutions for coordinating autonomic managers are based on software infrastructures, which are in charge of ensuring a global view of the managed system for all managers and synchronizing managers' operations.

However, coordination is a problem of synchronization and logical control of administration operations that can be applied by autonomic managers on the managed system at a given time in response to events observed on the state of this system. Therefore, its solution requires the use of models with events and states, where properties on the order of events or the mutual exclusion of parallel states can be addressed. Such models are at the basis of reactive or synchronous programming languages, and their compilation and analysis tools, as well as discrete control techniques.

D. *Our proposed approach*

Our approach is to consider the coordination as a synchronization management problem, and to design an additional layer, as shown in Figure 2, above the individual administration loops, which constitutes a coordination controller. This relies upon access to information about local controllers, such as their current state or execution mode, their controllable features (e.g., suspendability), and relevant events. We will build this hierarchical controller using models of reactive systems, which are automata-based, and Discrete Controller Synthesis to generate automatically the correct coordination constraint, so that logical coherence invariants are enforced.

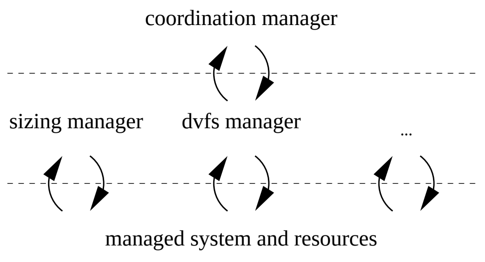


Figure 2. Coordination architecture of multiple loops

In this paper, we apply this approach to the case of the coordination of energy-aware controllers, which manage respectively Sizing (server provisioning) and Dvfs (processor frequency).

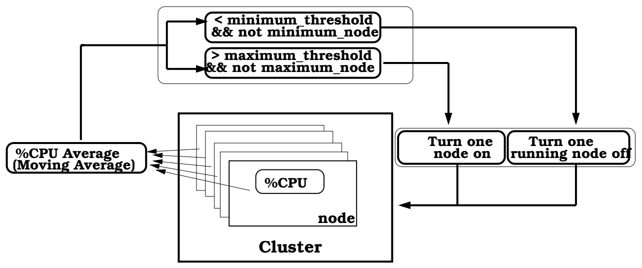

< minimum_threshold && not minimum_node
> maximum_threshold && not maximum_node
%CPU Average (Moving Average)
%CPU
node
Cluster
Turn one node on
Turn one running node off

Figure 3. Optimization controller

The rest of the paper is organized as follow. In section II, we present two energy-aware autonomic managers. In section III, we present the tools used in our approach for designing an efficient and correct coordination controller for autonomic managers. In section IV, we present the design of the coordination controller for the managers presented in Section II with our approach. In section V, we present a simulation of the generated coordination controller. Section VI presents the integration of the generated controller into a real system. In section VII, we discuss background and related work. Finally, in section VIII, we conclude the paper and outline directions for future work.

II. Uncoordinated control loops

We present two controllers dealing with energy optimization and performance of a system. They are developed independently. They try to optimize the energy consumption of a system while preserving a good performance. They are based on performance thresholds that describe an optimal performance region where the system must be depending on its workload.

A. *Optimization controller: Sizing*

This controller is for replicated servers based on a load balancer scheme. Its role is to dynamically adapt the degree of replication according to the system load. It dynamically turns cluster nodes on when the load of the system cannot be handled by resources it uses before the overload. When the system is underloaded, it turns cluster nodes off to save power under lighter load.

Figure 3 shows the execution scheme of the optimization controller. The controller analyzes the nodes CPU usage to detect if the system load is in the optimal performance region. It computes a moving average of collected load monitored by sensors. When the controller receives a notification from sensors, if the average exceeds the maximum threshold and the maximum number of replication (max node) is not reached, it increases the degree of replication by selecting one of the unused nodes. If the average is under the minimum threshold and the minimum number of replication is not reached, it decreases replication by turning a node off.

B. CPU-frequency controller: Dvfs

This controller targets single node management. Its role is to dynamically adapt the CPU-frequency of a node according to the load this node receives. It dynamically increases or decreases the CPU-frequency depending on the load.

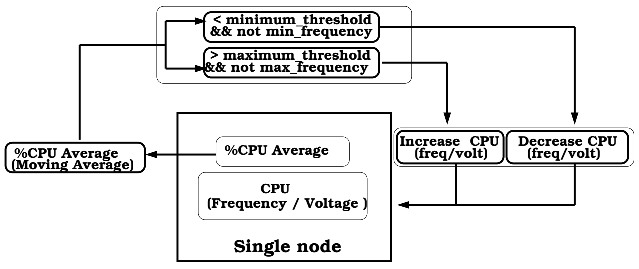

< minimum_threshold && not min_frequency
> maximum_threshold && not max_frequency
%CPU Average (Moving Average)
%CPU Average
CPU (Frequency / Voltage)
Single node
Increase CPU (freq/volt)
Decrease CPU (freq/volt)

Figure 4. CPU-frequency controller

Figure 4 shows the execution scheme of this controller. The controller analyzes the node CPU usage monitored by sensor. If the observed load exceeds the maximum threshold and the maximum CPU frequency is not reached, it increases the CPU frequency. If the load is under the minimum threshold and the minimum CPU frequency is not reached, it decreases the CPU frequency. This controller is local to the node it manages and is implemented either in hardware or software. The one we use is a user-space software and follows the on-demand policy.

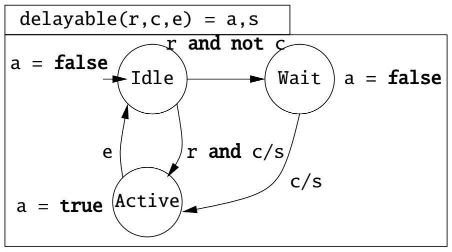
C. Uncoordinated execution

Here, we analyze the control of replicated servers composed of both controllers Sizing and Dvfs described above. Sizing deals with the whole system while Dvfs deals with each node separately.

When adding self-management capabilities to a system, one can use these two controllers to manage the energy consumption. The objective of using these two controllers together could be to optimize the energy consumption locally on each used node by acting on the CPU frequency and globally by managing the degree of replication. The objective is to optimize the energy consumption, without any coordination, however in some case this objective is not met. For example, when the system is overloaded, it is detected by Sizing and an upsizing operation is performed. But the system is overloaded means that some or all nodes that compose this system are overloaded, which implies CPU-frequency increase operation on nodes that are overloaded. If increasing the CPU frequency of these overloaded resources could be enough to restore the system performance to the optimal performance region, the upsizing operation become irrelevant, useless and leads to waste of energy since a new node is added while the previous nodes were able to handle the load received by the system after increasing the frequency of their CPU. There is a need to delay as long as possible upsizing operations when CPU-frequeny increase can be done.

III. Synchronous programming and discrete controller synthesis

For our contribution, we use the language BZR [3]. This language allows to describe reactive systems by means of generalized Moore machines, i.e., mixed synchronous dataflow equations and automata [11], with parallel and hierarchical composition. The basic behavior is that at each reaction step, values in the input flows are used in order to compute the values in the output flows for that step. Inside the nodes, this is expressed as a set of declarations, which takes the form of equations defining, for each output and local, the values that the flow takes, in terms of an expression on other flows, possibly using local flows and values computed in preceding steps (also known as state values).



```
node delayable(r,c,e:bool) returns (a,s:bool)
  let
    automaton
      state Idle
        do a = false ; s = r and c
        until r and c then Active
            | r and not c then Wait
      state Wait
        do a = false ; s = c
        until c then Active
      state Active
        do a = true ; s=false
        until e then Idle
     end
   tel
```

Figure 5. Delayable task in graphical and textual syntax.

Figure 5 shows a small program in this language. It programs the control of a task, which can either be idle or active. When it is idle, i.e., in the initial Idle state, then the occurrence of the input `r` *requests* the launch of the task. Another input `c` (which will be controlled further by the synthesized controller) can either allow the activation, or temporarily block the request and make the automaton go to a waiting state. When active, the task can end and go back to the idle state, upon the notification input `e`. This `delayable` node has two outputs, `a` representing activity of the task, and `s` being emitted on the instant when it becomes active : this latter is connected to the OS with the task starting operation.

The main feature of the BZR language is that its compilation involves *discrete controller synthesis* (DCS). DCS allows to compute automatically a controller, i.e., a function which will act on the initial program so as to enforce a given temporal property. Concretely, the BZR language allows the declaration of *controllable variables*, the value of which are not defined by the programmer. These free variables can be used in the program to describe choices between several transitions. These variables are then defined, in the final executable program, by the controller computed by DCS. DCS produces, when it exists, the maximally permissive constraint on the values of controllable variables, such that the resulting inhibited behavior satisfies the objective.

```
twotasks(r1, e1, r2, e2) = a1, s1, a2, s2
enforce not (a1 and a2)
with c1, c2

    (a1, s1) = delayable(r1, c1, e1)
    (a2, s2) = delayable(r2, c2, e2)
```

Figure 6. Mutual exclusion enforced by DCS in BZR.

Figure 6 shows an example of use of these controllable variables. This example consists in two instances of the `delayable` node, as defined in Figure 5. These instances run in parallel, defined by synchronous composition: one global step corresponds to one local step for every equation, i.e., here, for every instance of the automaton in the `delayable` node. Then, the `twotasks` node so defined is given a *contract* composed of two parts: the **with** part allowing the declaration of controllable variables (c_1 and c_2), and the **enforce** part allowing the programmer to assert the property to be enforced by DCS, using the controllable variables. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time. Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading automata of tasks to the wating state whenever the other task is active.

IV. MODEL-BASED COORDINATION

We propose a coordination solution, based on such reactive models, to avoid inconsistencies induced by these controllers running in parallel. This solution consists of designing a coordination controller on top of these controllers. This coordination controller is responsible of controlling the execution of Sizing and Dvfs in order to prevent any execution which may lead to inconsistencies.

The design of such a coordination controller is based on the synchronous approach. We use the synchronous programming to model the behavior of each controller. The models represent all the states in which they can be during their execution, with some control on transitions. The composition of these models describes the parallel execution of the controllers, which means both desired and undesired behaviors. We use discrete control synthesis techniques to automatically compute and generate the coordination controller based on the composition of the models and a coordination policy. The coordination policy is expressed as properties that should be enforced by the desired behaviors.

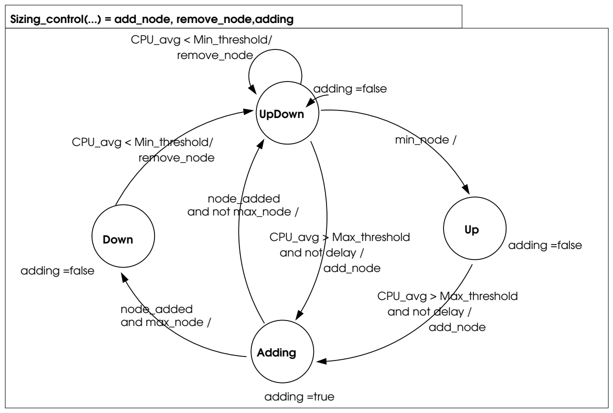


Figure 7. Optimization controller

A. *Optimization controller model*

The model of this controller is composed of two automata.

Figure 7 represents the automaton for Sizing, where we add a control of the upsizing operations. The control is represented by the Boolean **delay**, upsizing operations are possible only when this variable is false. The outputs of this automaton are three Booleans, **add_node** and **remove_node** being signals allowing the controller to request the resuming or suspension of a node, **adding** being true whenever an adding operation is performed.

The initial state is **UpDown**, both upsizing and downsizing operations are possible. When the CPU average reaches the maximum threshold and the upsizing operations are allowed, the controller requests a new node, and goes to the **Adding** state, where it awaits for the new node to be actually available. In this **Adding** state, nodes can neither be added nor removed. When node_added occurs, the controller can either go back to **UpDown**, or if there is no more nodes able to be resumed, go to the **Down** state where only downsizing operations can be performed. This **Down** state is left once one node is suspended. The **Up** state is used when no node can be removed, i.e., when the minimum number of replication is reached. In the **Up** state, only the upsizing operations can be applied.

Figure 8 represents how the Sizing manager can be controlled, by inhibiting add_node operations in some global states. The output of this automaton is the Boolean **delay**, which enables upsizing operations when it is false. This output feeds the input **delay** in Figure 7. Initially, the automaton is in the state **Idle** where upsizing operations are delayed. When **c** is true, it goes to the state **Active** where upsizing operations are allowed. It stays there until **c** is false.

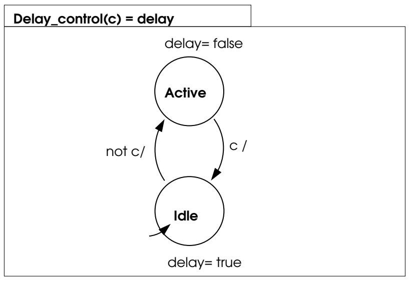


Figure 8. Upsizing operations control

B. CPU-frequency controller model

In our coordination problem, it is not necessary to control the execution of local dvfs controllers. We only need their current state in order to allow/deny upsizing operations. Therefore, a global observer is used for collecting information about current state of the set of Dvfs, each Dvfs provides two outputs, *min* being true when it can not decrease the CPU-frequency and *max* being true when it can not increase the CPU-frequency. This observer is a sensor that monitors the global state of set of local Dvfs. It has two outputs, one corresponding to the conjunction of all *min* outputs and the others to the conjunction all *max* outputs. Figure 9 represents the automaton for the observer. The outputs of this automaton are two booleans, **max_freq** being true when all local Dvfs reach the maximum frequency and **min_freq** being true when all local Dvfs reach the minimum frequency.

The inital state is **Normal** where **max_freq** and **min_freq** are false. In this state, at least one of the set of Dvfs can apply both CPU-frequency increase and CPU-frequency decrease operations. When all nodes are in their maximum CPU-frequency, the observer goes to the state **Max**. If all nodes are in their minimum CPU-frequency, the controller goes to the state **Min**. In the state **Max**, all Dvfs can only apply CPU-frequency decrease operations. In the state **Min**, they can only apply CPU-frequency increase operations.

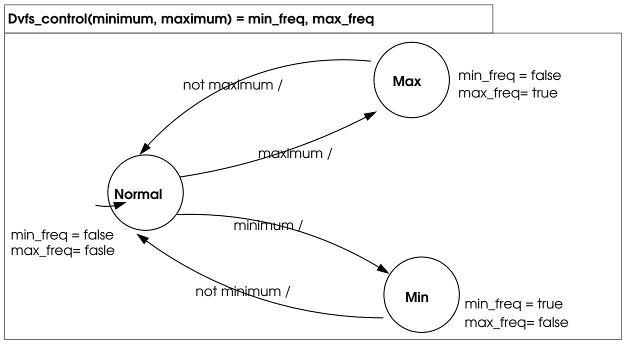


Figure 9. CPU-frequency controller

When the observer is in the state **Max** or the state **Min**, it stays there until at least one of the nodes is neither in its maximum CPU-frequency nor its minimum CPU-frequency.

C. Coordination policy

Finally, we present the coordination controller design.

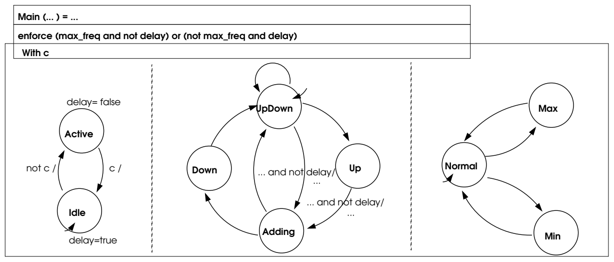


Figure 10. BZR program for coordination policy

Figure 10 is the coordination program built with the BZR language. The three automata presented before are composed in parallel, and a contract is added to enforce the coordination policy. Here, we want that the upsizing operations to be delayed when CPU-frequency increase operations can be performed. This coordination policy is stated by (max_freq **and not** delay) **or** (**not** max_freq **and** delay). The variable c is declared as a controllable.

V. Coordination controller simulation

The generated controller behavior can be simulated before its integration in the system with the SIM2CHRO chronogram tool (Verimag). It allows to test if the generated program reaction, represented by its outputs, respects the coordination policy expressed as logical invariant whatever the inputs are. Figure 11 represents a snapshot example of the complete simulation.

It shows a scenario illustrating the generated coordination controller in action. The input and output variables are Booleans. The input **minimum** notifies that all used nodes

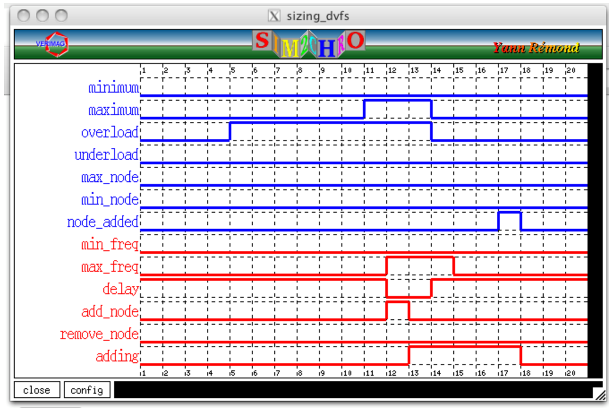


Figure 11. coordination controller simulation

are in their minimum CPU frequency and **maximum** notifies that all used nodes are in their maximum CPU frequency. The input **overload** represents the condition **CPU_avg > Max_threshold** and **underload** the condition **CPU_avg < Min_threshold**. **node_added** notifies that the previous adding node request have been treated succesfully. At the beginning, all used nodes are neither in their maximum CPU frequency nor in their minimum CPU frequency and the upsizing operations are not allowed (output **delay**).

At step 5, the input **overload** becomes true. This event should trigger an upsizing operation but, since all nodes are not in their maximum CPU frequency (output **max_freq**), this operation is not performed. An upsizing operation is performed only after the step 11 where all nodes are in their maximum CPU frequency.

VI. Implementation

A. Integrating the synchronous program into the system

The automata are composed in one BZR program. It is compiled and the generated code has two main functions: *reset* and *step*. The *reset* function allows to initialize the program and *step* to compute a reaction to events that correspond to the inputs of this function. The generated program is a reactive one. It has to be encapsulated into a loop that is responsible of executing the function *step* periodically or when an event occurs to get a reaction.

The coordination controller corresponds to the loop that encapsulates the BZR program. We have designed a program that is responsible of getting events from sensors for average load and Dvfs state, calling the function *step* with these events and transmitting the outputs of this function *step* to managers.

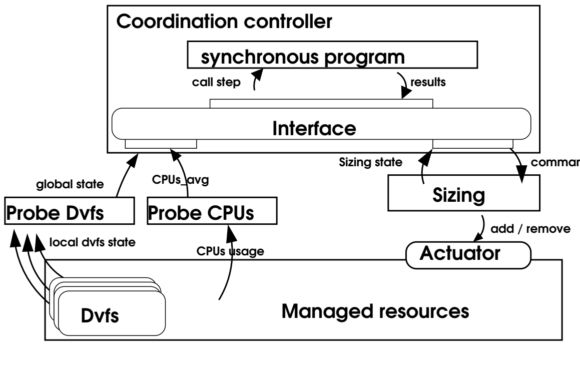


Figure 12. Integration

Figure 12 represents the architecture of a system in which the coordination controller is integrated. Since the role of this coordination controller is to control which manager should react or not to events, all sensed information is transmitted to the coordination controller instead of the managers. The outputs of the coordination, in reaction to events, are forwarded to the controlled managers i.e., in this case the manager Sizing. The interface allows interaction between the synchronous program, the sensors and the manager.

B. Connecting the automata

The automata are connected to the system by its input events, and by outputs which are commands to be applied in the system.

The automata represent the current state of a part of the system, which the coordination controller needs in order to make a decision when events occur. The inputs of automata have to be fed with events occurred in the system for making them evolve and their outputs have to be applied to the system for acting on its state.

The automaton *Dvfs_control* informs about the global state of the set of local Dvfs and its outputs serve only for the decision the controller has to make. The inputs of this automaton correspond to the outputs of the probe Dvfs. The automaton *Sizing_control* manages Sizing execution. It describes the current state of Sizing and decision it takes in response to events. The input **CPU_avg** corresponds to the average of system load. The inputs **max_node** and **min_node** correspond to the capability for Sizing to apply operations, max_node informing about the capability to perform an upsizing operation and min_node a downsizing operation. The output **add_node** respectively **remove_node** are triggering the operations Sizing performs when **CPU_avg** is over **Max_threshold** respectively **CPU_avg** is under **Min_threshold**. **add_node** being true respectively **remove_node** being true means Sizing has to add a new node respectively remove a node. The automaton *Delay_control* has one input which is managed by the generated controller. Its output is used as input for the automaton *Sizing_control*, in order to control some transitions Sizing_control can take.

C. Implementation architecture

This approach has been implemented for the management of a clustered web server. The managed system consists of one server Apache and replicated servers Tomcat.

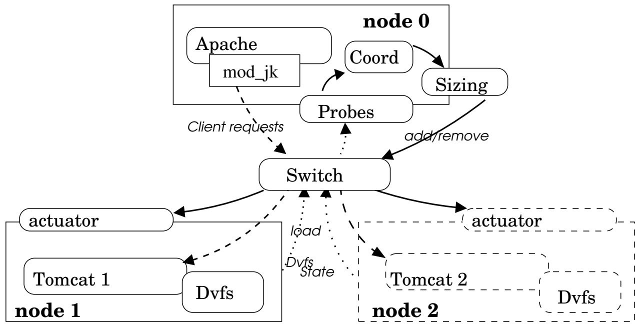


Figure 13. Experimental platform: architecture

The experimental platform, as shown in Figure 13, consists of a network of three nodes. Node 0 hosts the Apache server, each of node 1 and node 2, one Tomcat server. Initially, only one Tomcat server is active. Both Tomcat servers are active when the requests received cannot be handled by one. Unlike to the execution without coordination, where undesired behaviors have been observed, we observe that the coordination execution follows the defined policy. Upsizing operations are performed only when all active nodes hosting a server Tomcat are in their maximum CPU frequency.

In order for this controller to work well, it is important that it runs sufficiently frequently compared to the load dynamics : for every load peak to be detected and managed, the frequency of sampling and the communication must be fast enough.

VII. Related work

Concerning energy control, many works addressed energy management on datacenters. Some of these researches are based on (i) hardware with voltage and frequency control (e.g., DVFS [6]), (ii) resource allocation: Reducing power consumption by reducing the clock frequency of the processor has been widely studied [7] [18], Flautner et al. [5] explored a software managed dynamic voltage scaling policy that sets CPU speed on a task basis rather than by time intervals. [4] proposes a power budget guided job scheduling policy that maximizes overall job performance for a given power budget. [1] [13] [14] focused on dynamic resource provisioning in response to dynamic workload changes. These techniques monitor workloads or other SLA (Service Level Agreement) metrics experienced by a server and adjust the instantaneous resources available to the server. Depending on the granularity of the server (single or replicated), the dynamically provisioned resources can be a whole machine in the case of replicated servers. Energy efficiency is achieved using a workload-aware, just-right dynamic provisioning mechanism and the ability to power down subsystems of a host system that are not required.

While these works are relevant, they did not adress the problem of coordinating multiple energy managers. Our work is complementary since it can be used to build a system that includes more that one of the previous approaches. Few works have also investigated manager coordination for energy efficiency. Kumar [10] proposes vManage, a coordination approach that loosely couples platform and virtualization management to improve energy savings and QoS while reducing VM migrations. Kephart [2] addresses the coordination of multiple autonomic managers for power/performance tradeoffs based on a utility function in a non-virtualized environment. Nathuji [12] proposes VirtualPower to control the coordination among virtual machines to reduce the power consumption. These works involve coordination between control loops, but these loops are applied to the managed applications. However, these work propose adhoc specific solutions that have to be implemented by hand. If new managers have to be added in the system the whole coordination manager need to be redesigned.

In contrast with [15], which relies on formal specification to derive a formal model that is guaranteed to be equivalent to the requirements, our work can be related to the applications of control theory to autonomic or adaptive computing systems [8]. In particular, Discrete Event Systems in the form of Petri nets models and control have been used for deadlock avoidance problems [17]. Compared to these works, we rely on synchronous programming and discrete controller synthesis. Once an autonomic manager is modeled as automata, inserting this autonomic manager with other pre-existing just require to update the coordination invariants. The new coordination manager is automatically generated from the managers models and the coordination invariants. In contrast with [16], which addresses the management of datacenters based on thermal awarness with external sensing infrastructure for energy and cooling efficiency, the work, presented in this paper, focuses on coordinating multiple workload-aware managers to ensure an energy efficiency.

VIII. Conclusion and future work

One major challenge in system administration is the coordination of multiple autonomic managers for correct and coherent administration. In this paper we presented an approach for coordinating multiple self-management modules in a consistent manner to manage a system. This approach, based on synchronous programming and Discrete Controller Synthesis, has the advantage of generating the required controller to enable the correct by construction coordination of multiple autonomic managers. The advantages of this approach are following:

- High-level of programming
- Correctness of the controller
- Automated generation/synthesis of the controller
- That is maximally permissive

We have tested this approach for coordinating two energy-based self-management modules: Sizing, which manages the degree of replication for a system based on a load balancer scheme, and Dvfs, which manages the level of CPU frequency for a single node. In this case, the coordination policy was to allow Sizing to add new node only when all Dvfs modules cannot apply increase operations at all in response to the increasing load the system receives.

For future work, we plan to evaluate this approach for large scale coordination with more complex coordination policies and several managers, combining both self-optimization and self-regulation frequency managers with self-repair manager that heal fail-stop clustered multi-tiers system.

Acknowledgement

This research is supported by ANR INFRA under a grant for the project ctrl-Green.

References

[1] Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *Cluster Computing*, 2006.

[2] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan. Autonomic multi-agent management of power and performance in data centers. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track*, AAMAS '08, pages 107–114, Richland, SC, 2008.

[3] Gwenaël Delaval, Hervé Marchand, and Eric Rutten. Contracts for modular discrete controller synthesis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 57–66, 2010.

[4] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Optimizing job performance under a given power constraint in hpc centers. In *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, pages 257–267, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. *Wirel. Netw.*, 8:507–520, September 2002.

[6] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[7] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MOBICOM'95*, pages 13–25, 1995.

[8] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.

[9] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[10] Sanjay Kumar, Vanish Talwar, Vibhore Kumar, Parthasarathy Ranganathan, and Karsten Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 127–136, New York, NY, USA, 2009. ACM.

[11] Jean louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *In ACM International Conference on Embedded Software (EMSOFT' 05*, pages 173–182. ACM Press, 2005.

[12] Ripal Nathuji and Karsten Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 265–278, New York, NY, USA, 2007. ACM.

[13] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems, 2001.

[14] Ivan Rodero, Juan Jaramillo, Andres Quiroz, Manish Parashar, Francesc Guim, and Stephen Poole. Energy-efficient application-aware online provisioning for virtualized clouds and data centers. pages 31–45, 2010.

[15] Roy Sterritt, Michael Hinchey, James Rash, Walt Truszkowski, Christopher Rouff, and Denis Gracanin. Towards Formal Specification and Generation of Autonomic Policies. In *Embedded and Ubiquitous Computing*, pages 1245–1254, 2005.

[16] Hariharasudhan Viswanathan, Eun Lee, and Dario Pompili. Self-organizing sensing infrastructure for autonomic management of green datacenters. *Ieee Network*, 25(4):34–40, 2011.

[17] Yin Wang, Terence Kelly, and Stéphane Lafortune. Discrete control for safe execution of it automation workflows. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 305–314, New York, NY, USA, 2007. ACM.

[18] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.