Automatic Configuration of an Autonomic Controller: An Experimental Study with Zero-Configuration Policies

Thomas Heinis[1], Cesare Pautasso[2]
[1]Systems Group, Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland
[2]Faculty of Informatics, University of Lugano, 6900 Lugano, Switzerland
heinist@inf.ethz.ch, cesare.pautasso@unisi.ch

Abstract

Autonomic control managers can remove the need for manual system configuration in order to achieve good performance and efficient resource utilization. However, simple controllers based on reconfiguration actions tied to thresholds, or 'if-then' rules, themselves need to be configured and tuned in order to adapt the controller behavior to the expected workload characteristic. In this paper we present an experimental study of zero-configuration policies that can be automatically tuned based on analytical models of the system under control. In particular, we have designed and implemented a threshold-free self-configuration policy for a distributed workflow execution engine and compared it with a standard PID controller. The experimental results included in the paper show that using such a policy the controller can tune itself in addition to reconfiguring the distributed engine and the proposed policy out-performs simpler policies that require manual and error-prone tuning of their parameters.

1. Introduction

Automatic self-configuration is one of the most important properties of autonomic systems [9]. In order to remove the need for manual configuration, an autonomic controller is usually attached to a system to monitor its operation and automatically apply reconfiguration actions [14]. However, this is only a partial solution to the self-configuration problem if the autonomic controller itself requires to be properly configured and tuned. If it is difficult to set the optimal configuration of a system, but at least human system administrators have gained some experience dealing with it, optimally configuring an autonomic controller is an even harder problem [3]. As an example, in our experience with the JOpera autonomic workflow engine [6], we observed a 287% performance variation depending on the values of thresholds used by the autonomic controller policies, which is similar to the performance variation of a misconfigured engine.

In this paper we argue that autonomic controllers that require to be configured – with parameters, thresholds, or 'if-then' rules – defeat the goal of automatic self-configuration and do not really help system administrators to deal with the complexity of managing their systems [10]. Instead, they potentially make it even more complex to manage a system, due to the need of understanding the impact of the controller configuration parameters on the overall system performance.

As a first step to address this problem, we looked at standardized controllers (such as the PID controller [7]), for which specific tuning techniques are available to deal with the configuration of the controller. However, if the characteristics of the system or the workload change, the PID controller may need to be tuned again. In this paper, we compare this standard approach with a *zero-configuration controller* and show how it can be built starting from an analytical model of the system under control. Our approach is applicable to stage-based architectures [17], where a variable number of processing stages are loosely connected by queues. The controller employs a balancing configuration policy, which strives to balance the rate of message production and consumption by determining the optimal number of stage replicas around a certain queue. This policy does not depend on any threshold and it can be automatically tuned based on observable parameters of the system. In order to give an experimental comparison of the two approaches, we have implemented both kinds of controllers for the JOpera autonomic workflow engine [6]. We then conducted an extensive evaluation study using real workloads. In this paper we present our first results showing the feasibility and benefits of zero-configuration control policies in a practical setting.

The remainder of this paper is structured as follows. In Section 2 we first motivate our work and provide the context for it in Section 3. We then describe the implementa-

tion of a PID Controller policy in Section 4 as baseline and then present our zero-configuration policy in Section 5. In Section 6 we compare the different policies, present related work in Section 7 and draw conclusions in Section 8.

2. Motivation

We begin with an experiment to show the significant impact that the configuration of an autonomic controller may have on the performance of an autonomic system. In particular, we have assigned different values to two thresholds used by the best-known policy of the controller of the JOpera autonomic workflow engine [6]. This Growth policy monitors the sizes of two different queues and uses these as an indication of how much work needs to be processed by each component of the engine. If the growth in either of the queues exceeds predefined thresholds, then the system will be reconfigured in order to devote more resources to the component consuming the growing queue.

As Figure 1 clearly illustrates[1], the time required to execute the same workload is highly sensitive to the threshold settings of the autonomic control policies. If thresholds are not set optimally, the performance of the system will suffer (by 287% in the worst case). Also, the relationship between the performance and the thresholds is non-linear, making it difficult to find the optimal threshold values. Setting higher thresholds makes the autonomic system slower to adapt and thus also slower to execute its workload, as it will take longer to reach the appropriate configuration. Reducing the threshold values makes the system change its configuration more often. If the thresholds become too small, the reconfiguration overhead also noticeably affects the system's performance.

This problem can be approached in different ways: Either an automatic way of setting such thresholds is found or a policy that does not require thresholds is developed.

3. Autonomic Workflow Execution

To put our study in context, in this section we outline the architecture of the JOpera autonomic workflow engine. For more information, we refer the interested reader to [6, 15, 16]. We have chosen JOpera for this experimental study because it offers an extensible research platform where new control policies can be easily plugged into the controller and their performance can be compared and benchmarked using real workloads.

[1]For the experimental setup and the workload used, please refer to Section 6.1 and 6.2 respectively.

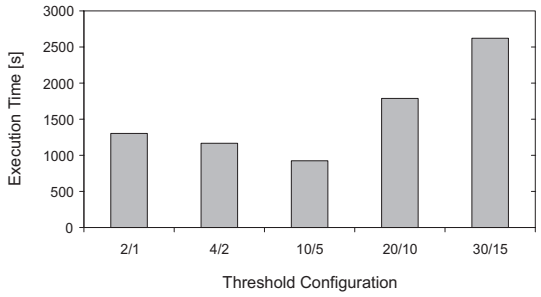

Figure 1. Impact of threshold configuration on the autonomic system performance

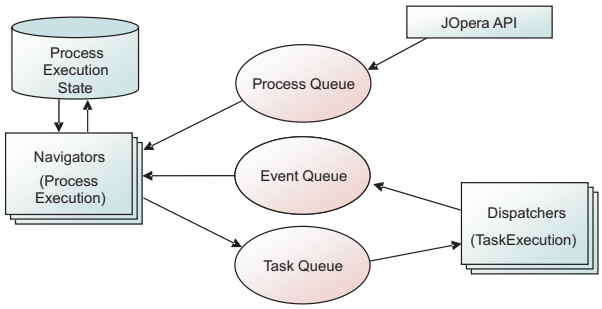

Figure 2. Workflow Engine Architecture

3.1. Workflow Engine

JOpera [1] is a rapid service composition tool offering a rich visual environment built on top of Eclipse to support the whole lifecycle of workflow modeling and execution. In this paper we focus on the autonomic capabilities of the JOpera distributed workflow engine, which can be deployed over a cluster of computers to handle a large number of concurrent workflow executions.

As shown in Figure 2, the engine is designed using a stage-based architecture. In particular, the workflow execution (or *navigation*) stage is decoupled from the invocation (or *dispatching*) of the individual tasks of the workflow. The navigator and the dispatcher stages communicate asynchronously using message queues.

Workflow execution is initiated through the engine API, which posts a new process execution request message into the *Process* queue. This request is processed by the navigator, which instantiates a new workflow and begins to determine which of its tasks should be executed next. Once it has done so, it puts task execution requests into the *Task* queue. These messages are received by the dispatchers that carry out the actual task execution. Dispatchers use the *Event* queue to inform navigators about the progress and the completion of the task execution. Navigators also con-

sume messages from the *Event* queue to update the state of the workflow processes with the results of the execution of the tasks.

This architecture for distributing the workflow execution has the advantage that workflow execution is not stalled due to a possibly long-running task. Multiple workflows having parallel tasks can be concurrently executed by allocating a pool of threads to perform the corresponding navigation and dispatching. Also, by providing an appropriate implementation of the message queues, it is possible to scale-out this architecture over a cluster of computers, where each node is allocated to run a navigator or a dispatcher stage.

3.2. Autonomic Controller

In our previous work we have added an autonomic controller component to the workflow engine to automatically determine the allocation of the cluster to navigator and dispatchers. This is very important, as the choice of how to partition the cluster between the two stages can heavily influence the performance of the engine. In one experiment [6], the penalty for a misconfigured engine reached 533% in the worst case. Also, we have shown that the optimal allocation depends on the characteristics of the workload, and thus the system should be continuously monitored and adapted to its workload.

To do so, the autonomic controller runs on a separate node of the cluster. It monitors the engine's performance and reconfigures it according to specific control strategies. The control loop for self-configuration is shown in Figure 3. First the controller measures the system's performance according to the information strategy. Based on the new information, it decides whether a reconfiguration is needed using the optimization strategy. The configuration changes are implemented by choosing which nodes of the cluster will be affected based on a selection strategy. It is worth noting that it may not always be possible to apply all reconfiguration decisions, as these are constrained by the amount of available resources. After a change has been applied, its effects may not be immediately visible. Hence, to avoid repeating the same decisions based on out of date information, the controller waits for changes to take effect before restarting the loop.

The *selection strategy* defines which nodes are chosen to be reconfigured, e.g., it prioritizes idle nodes over busy ones and can also predict the cost of stopping and migrating an active navigator/dispatcher based on how many workflows/tasks are being executed on a particular node.

The *information strategy* defines which performance indicators of the engine are monitored (e.g., queue sizes, queue growth, execution throughput etc.). An additional benefit of stage-based architectures is that the resulting system can be easily instrumented to gather useful performance

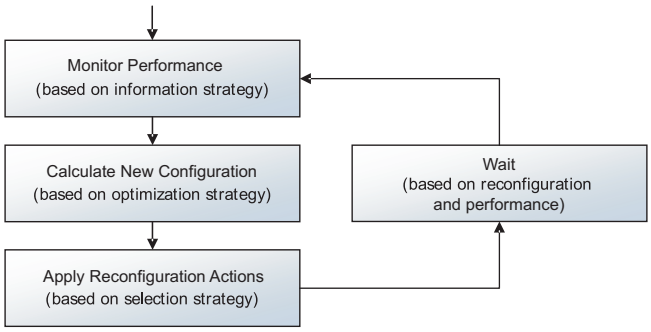

Figure 3. Steps taken by the controller

information by observing the behavior of its queues and measuring the rate at which messages are processed by each stage of the engine. Updates of each of these indicators are periodically sent by each node of the cluster to the autonomic controller using a *statistics* queue.

The *optimization* strategy maps the information collected into reconfiguration actions. To do so, the controller may use different policies that can be easily replaced to be compared. In [15] we have introduced simple rules that detect imbalances in the system by comparing the size of a queue with some thresholds. Whereas this is enough to reconfigure the system automatically in response to workload changes, in the rest of this paper we propose more advanced optimization policies that do not require thresholds to do so. In the context of this paper we will focus on improving the optimization strategy while the selection and the information strategy will remain as described before.

4. PID Controller

As a baseline for our empirical comparison, in this section we introduce a standard PID Controller used to implement the optimization policy of the autonomic controller.

A PID controller [7] is a common feedback loop used in many traditional industrial control systems. As shown in Figure 4, it maps the control error ($e(t)$, which measures how far the system is from the reference input) to a control action that aims at correcting such error. To do so, it combines three terms: proportional, integral, and derivative, each having its own weight (or gain).

$$c(t) = c_p e(t) + c_i \int_0^t e(\tau)d\tau + c_d \frac{de(t)}{dt} \qquad (1)$$

The proportional part corrects the current error, the integral part compensates for the steady state error (if $e(t) = 0$) and the derivative part helps to avoid oscillations.

To apply the PID controller to the JOpera engine, we need to define the control error in terms of the engine's per-

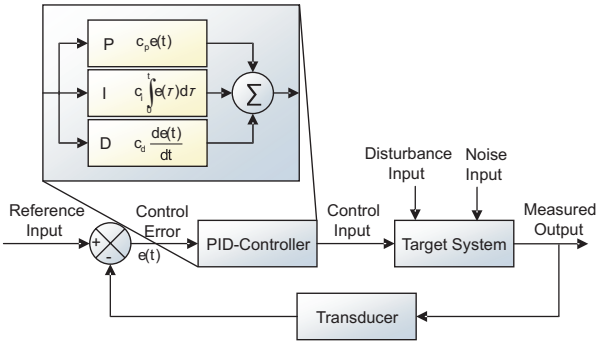

Figure 4. Feedback control loop with PID controller

formance and define the control actions in terms of which reconfiguration actions are available.

Since we are interested in removing all external dependencies of the controller, we choose not to rely on an external input defining the set-point of the system (which would have to be adjusted by system administrators). Instead, we rely on a combination of observable internal parameters (i.e., the size of the queues) only.

Based on the intuition that the system is ideally configured if its queues are of equal size, we combine the measurements of the queue sizes as follows:

$$E(t) = \frac{q_{Process}(t) + q_{Event}(t)}{q_{Task}(t)} \tag{2}$$

where $q_{Queue}(t)$ is the current queue size. For $q_{Task}(t) = 0$ we set $E(t) = \infty$.

Since $E(t) > 0$, it is not suitable to be used as direct input to the PID controller. We thus normalize it:

$$e(t) = \begin{cases} E(t) - 1 & E(t) \geq 1 \\ 1 - \frac{1}{E(t)} & 0 < E(t) < 1 \end{cases} \tag{3}$$

With this definition of the control error, the system is balanced if $q_{Process}(t) + q_{Event}(t) = q_{Task}(t) \Rightarrow e(t) = 0$ and thus, no reconfiguration action should occur. Because in this case the control input is zero (as no change in the configuration is required), the steady state error is zero as well and the integral term can therefore be dropped ($c_i = 0$) from the PID controller.

Likewise, we can define what reconfiguration action should be taken. If $q_{Process}(t) + q_{Event}(t) < q_{Task}(t) \Rightarrow e(t) < 0$, more dispatchers should be added, as the task queue is larger. If $e(t) > 0$ we are in the opposite situation, and more navigators (and less dispatchers) are required. The resulting abstract control error in the interval of $[-\infty, \infty]$ is then mapped by the selection strategy to an actual reconfiguration in the interval bounded by $[0, a]$, where a is the total number of nodes available in the cluster.

The resulting PID controller still needs to be tuned by setting appropriate values to the gains of the proportional c_p and derivative c_d terms. The advantage of choosing a standard controller is that the problem of tuning its parameter is well understood [18] and several heuristics are available [5]. Some of these, however, require to subject the system to controlled input waveforms and cannot always be applied to tune a system in production which may be subject to unpredictable workloads. The resulting PID controller is also not robust with respect to changes in the workload, and thus has to be tuned repeatedly. Moreover, to rely on a PID controller with a single control error input we had to combine multiple measurements of the system's performance in a somewhat arbitrary way. Using a Multiple-Input Multiple-Output PID controller would have made its automatic tuning more difficult [12]. We expect to obtain better results with a controller that is based on a model more tightly coupled to the architecture of the system under control.

5. Balancing Zero-Configuration Policy

The motivation for this policy is to go beyond the fairly simplistic approach used by the Growth policy (Section 2) and the very general solution of the PID Controller as discussed in the previous Section 4. The goal of this policy is to use an analytical model which is more tightly coupled to the characteristics of the system and which, as the PID controller policy does, refrains from using thresholds.

This policy is referred to as the *Balancing policy* as it tries to balance the consumers and producers of the queues based on the rate at which messages are written and read from them.

Figure 5 shows a summary of the produced and consumed messages. A navigator takes r_n messages per second from the event queue. Each message is processed and, depending on the structure of the workflow being executed, a task execution request may be written to the task queue with the probability c_1. Therefore – assuming that all navigators work at the same rate – in one second n navigators take nr_n messages from the event queue and enqueue $c_1 nr_n$ tasks into the task queue. External process execution requests get into the system via the process queue with rate e.

On the other side of the queues, a dispatcher takes r_d task execution request messages per second from the task queue and sends, for each executed task, c_2 messages back into the event queue. Therefore – again, assuming a uniform task execution speed – d dispatchers get dr_d tasks from the task queue and write $c_2 dr_d$ events every second to the event queue.

The queue growth is the difference of the rate at which messages are put into the queue and are taken out of it. We

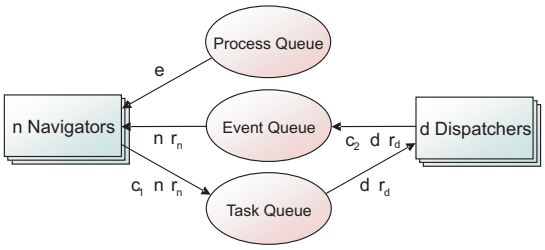

Figure 5. Modeling Communication Flows through the Queues

define p as the total number of messages in the queues consumed by the navigators (the event and process queue) and q as the number of messages in the task queue, consumed by the dispatchers. Hence the growth of the queues (p' and q') can be defined as follows, taking into account the number of messages that are written and read from a queue per unit of time.

$$p' = c_2 d r_d + e - n r_n \quad (4)$$

$$q' = c_1 n r_n - d r_d \quad (5)$$

Following the same strategy as with the PID controller, the goal of this policy is also to ensure equal queue growth, so that $p' = q'$:

$$(c_2 + 1) d r_d + e = (c_1 + 1) n r_n \quad (6)$$

We resolve (6) with respect to n:

$$n_{opt} = \frac{(c_2 + 1) d r_d + e}{(c_1 + 1) r_n} \quad (7)$$

This equation represents how many navigators (n_{opt}) are needed in order to obtain the balanced state $p' = q'$.

If we express the number of dispatchers d as the difference between the number of available nodes in the cluster a and the number of navigators n, we can substitute $d = a - n$ and can define the balanced configuration as a function of measurable variables:

$$n_{opt} = \frac{(c_2 + 1) a r_d + e}{(c_2 + 1) r_d + (c_1 + 1) r_n} \quad (8)$$

In order to calculate (8), the controller continuously averages the execution rates of navigators and dispatchers to arrive at r_d, r_n, measures e from the process queue and uses a from the current configuration of the cluster. The values of the two parameters c_1 and c_2 are calculated by solving (5) and (4):

$$c_1 = \frac{q' + d r_d}{n r_n} \quad (9)$$

$$c_2 = \frac{p' + n r_n - e}{d r_d} \quad (10)$$

and by additionally measuring the growth in both queues (p', q').

As opposed to the information fed into the PID controller, the balancing policy depends on more information (i.e., the rates e, r_d, r_n, p', q') but does not require any manual tuning.

In order to prevent the system from behaving erratically at startup, the values of the parameters c_1, c_2 can be initialized by analyzing the communication protocol between the dispatcher and navigator. The constant c_2 is the number of messages sent by the dispatcher into the event queue during the execution of a task. By design $c_2 = 4$, as the dispatcher goes through four different states during task execution and sends a notification for each state transition. The constant c_1 is defined as the probability the navigator sends a message into the task queue when it processes an incoming message. This depends on the structure of the workflow being executed. However, in general, for a workflow of t tasks, the navigator will receive $4t$ events from the dispatcher, one from the API and 5 from the state changes of the process executed. Thus:

$$c_1 = \frac{t}{4t + 6} \quad (11)$$

Since a process contains at least on task ($t = 1$), the lower bound of c_1 is $\frac{1}{10}$. For the upper bound:

$$\lim_{t \to \infty} \frac{t}{4t + 6} = \frac{1}{4} \quad (12)$$

From this analysis, the initial value of c_1 should be bound to the interval $[\frac{1}{10}, \frac{1}{4}[$.

To validate this analysis we have measured the values of the two constants using a heterogeneous set of workflow patterns. Figures 6 and 7 show that the observed values of the parameters are consistent with the analysis across all the kinds of workflows. As an interesting result, the standard deviation of c_1 for workloads with parallel processes is much higher than for other workloads. This is caused by the uneven distribution of tasks written to the task queue during process execution: All tasks of a parallel process are written to the queue at the beginning of the process and none thereafter.

6. Evaluation

In this section we evaluate the performance of the autonomic workflow engine with a controller that uses the policies presented in Sections 4 and 5. We analyze the behavior of the Balancing policy and the PID Controller policy for three different workloads. We compare the workload execution durations for the different policies, including – as a baseline – the Growth policy (Section 2) that requires manual setting of the thresholds.

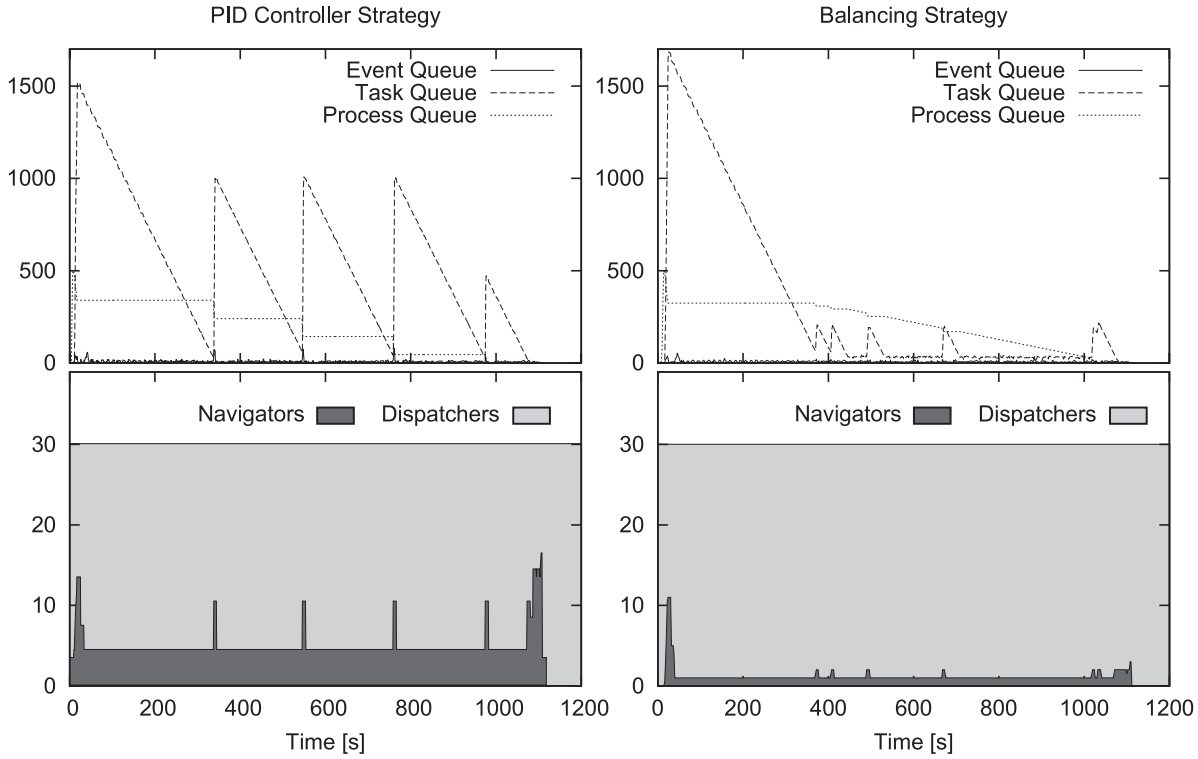

Figure 8. Busy Workload Execution Plots

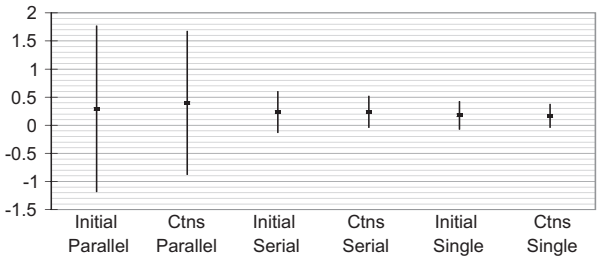

Figure 6. Measuring c_1 over multiple workloads

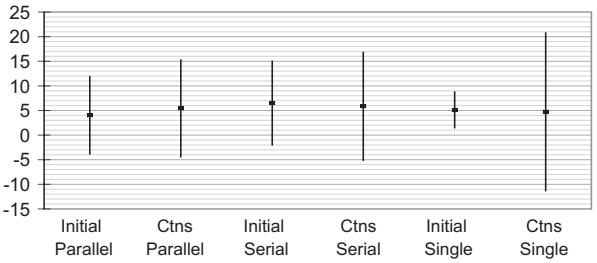

Figure 7. Measuring c_2 over multiple workloads

6.1. Experimental Setup

The experiments were carried out using two clusters. The first cluster consists of 30 1GHz dual P-III with 1GB memory, running Sun's Java Development Kit version 1.5.09 on Linux kernel version 2.4.22. The second cluster consists of six 1.4GHz dual AMD Athlon with 1GB memory, running JDK version 1.5.09 on Linux kernel version 2.6.8. The autonomic controller was run by an additional dedicated node.

6.2. Busy Workload

The busy workload consists of 500 workflows, each executing in parallel 10 CPU intensive tasks performing number primality tests with an execution time of 10 seconds. All workflows are started at the beginning of the benchmark. Because of the resource consumption and long duration of the tasks, we expect this workload to saturate all available nodes of the cluster and thus require as many dispatchers as possible to execute the tasks.

Figure 8 shows the execution plots for the Balancing policy and the PID Controller policy. The upper charts show

the evolution of the process, event and task queue, while the lower charts show the evolution of the configuration consisting of the number of navigators and dispatchers allocated.

The PID Controller policy adapts quickly to the growing task queue and allocates a maximum of 29 dispatchers for the whole execution duration. Once the task queue is nearly empty, the control error will lead the controller to start a number of navigators. These additional navigators will quickly fill up the task queue due to the structure of the workflows having 10 parallel initial tasks. The growth of the task queue will cause the controller to increase the number of dispatchers. This pattern is repeated several times, letting the configuration oscillate until the workload is processed completely.

The Balancing policy behaves very similar to the PID Controller policy in that it assigns most nodes of the cluster to run dispatchers. In contrast to the PID controller it avoids oscillations, resulting in a much smoother evolution of the configuration. The stability of the configuration also results in a steady decrease of the process queue size.

6.3. Burst Workload

The motivation of this workload is to test whether the autonomic controller can reconfigure the system as the characteristics of the workload change. The workload starts with a burst of 500 processes which execute a sequence of 10 CPU intensive tasks lasting 1 second. As soon as 95% of the processes terminate, the second burst is started. It consists of 2000 processes of 10 parallel empty tasks with a duration approximately 0s. Again, as soon as 95% of the processes have finished, a third burst is started similar to the first one. The fourth burst has the same characteristics as the second.

We expect an approximation of the following configuration evolution for this experiment. When bursts 2 & 4, each containing ten tasks to be executed in parallel are fed into the system, the task queue will grow very fast. As the tasks however can be executed in virtually no time, not many dispatchers will be required. The event queue will soon after also start to grow very fast. This is due to the many events that are generated for the quickly executed tasks. Because of the proportionally big event queue, the controller is likely to allocate more navigators. In contrast to this, the task queue will not grow nearly as quickly in case of bursts 1 & 3. In fact, the task queue is expected to grow shortly and then to maintain the size (as for each task taken from it, a new task will be enqueued as the tasks are executed serially). After each burst we expect the controller to slowly increase the number of dispatchers as this type of workload requires more dispatchers. The evolution of the queues sizes described before can be observed in Figure 9.

The PID Controller policy does not completely follow the expected reconfigurations. It starts with about 20 dispatchers in order to cope with the more CPU intense workload. At second 170, after the second burst has been started, it starts to allocate more navigators. The reconfiguration required for the third burst however is not completely carried out. The controller allocates roughly equal numbers of navigators and dispatchers. The configuration change for the fourth burst again is as expected as more navigators are allocated.

The Balancing policy behaves according to the expectations outlined before by allocating a large number of dispatchers at the first and the second burst. For the second burst the number of navigators is increased but the configuration is unstable. It seems to be much harder to get a stable configuration for processes with longer and busier tasks than with short idle tasks. The reason for this is the high execution rate of the navigators. A navigator is able to handle up to 200 events per second. Stopping or starting just one navigator results in a big change in the event queue growth which then leads to a very varying input for the policy. The fourth burst finally is handled by using about 20 navigators.

Besides a difference in the reconfiguration decisions taken in response to the bursts, the Balancing policy also seems to maintain a much more stable configuration throughout the experiment. In contrast, the controller performs many more reconfigurations and does not get into a stable state in case of the PID Controller policy. Moreover, the configuration seems to oscillate at times, for instance during the execution of the first burst. This oscillation is due to the parameter setting of the PID controller for this specific type of workload. The parameters are set once and are used for all workloads. It clearly is not possible to choose these values so that they fit for all types of workloads. During the whole course of the experiment, the PID Controller policy requires twice as many reconfigurations than the Balancing policy does, performing approximately 200 reconfigurations.

6.4. fMRI Workload

The processes of this workload are used in real experiments in the field of Functional Magnetic Resonance Imaging (fMRI)[8]. Such a process is used to process raw data of brain scans and takes in a first step the raw data, aligns it to a reference brain image by reslicing it, averages over several scans executed with different wavelengths, and finally slices along the x, y and z dimensions. The structure is simple, with two phases of parallel program executions.

The workload consists of ten fMRI processes which are started one after the other with a delay of 10s. The challenge of this workload is that the processes contain relatively long executing tasks (up to twenty seconds). As the workload consists only of few processes started continuously one after the other, the queues are virtually empty for most of the

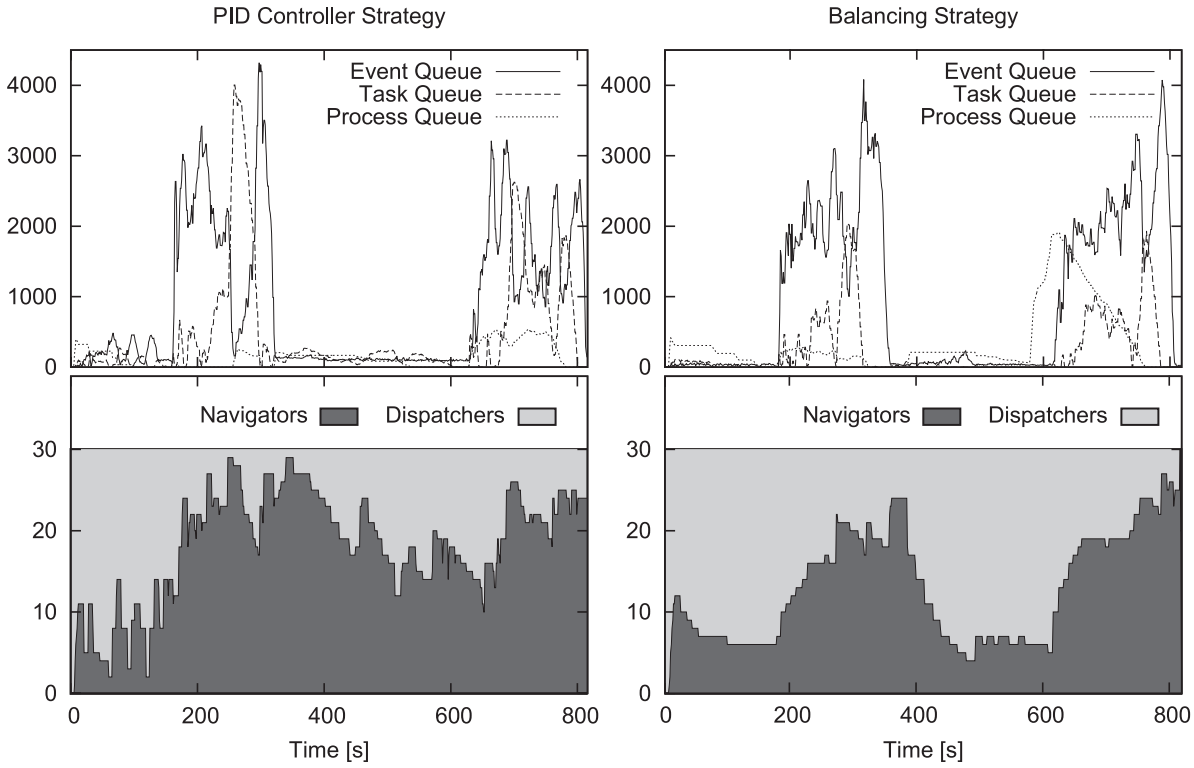

Figure 9. Burst Workload Execution Plots

time (see plots in Figure 10). This makes it very difficult for a policy observing queue sizes to select an appropriate configuration.

The PID Controller policy reacts to this challenging workload with many reconfigurations. As the task queue is mostly empty and the event queue shows few peaks only, this policy assigns on average slightly more navigators than dispatchers. A spike in the event queue is usually followed by an allocation of an increased number of navigators. Again in case of this workload, the PID Controller policy seems unable to maintain a stable configuration, letting the configuration oscillate slightly.

The Balancing policy behaves better than the PID Controller policy and on average uses two dispatchers more, which seems reasonable in face of the computationally very intense tasks. This policy also leads to fewer reconfigurations (about 20 compared to 40 reconfigurations for the PID Controller policy as Figure 12 shows).

6.5. Comparison

Figure 11 shows the execution durations of the three workloads using the Growth policy, the PID Controller policy and the Balancing policy. For the Growth policy, the best known threshold settings (10/5) were used.

As the qualitative analysis already has pointed it out, the the PID Controller policy and the balancing policy are about equally fast (average of 1103.5s and 1098.3s) for the Busy Workload, but more than 40% faster than the Growth Strategy (1920s). The Growth policy does not assign as many dispatchers as the other two policies do and hence requires more time to execute the workload.

For the Burst Workload the improvement is not as explicit, but still 20% for the Balancing policy (738.45s) and 9% for the PID Controller policy (837.88s) compared to the Growth policy (917.98s). This is still a good result, even if the qualitative analysis has shown that improvements are still possible. If we change the constants of the Growth policy, we are able to push down the execution time to the value of the PID Controller, but we loose about 20% performance in case of the Busy Workload, illustrating again that thresholds are geared toward specific workloads.

In case of the fMRI Workload, we have about the same performance gain than for the Burst Workload. The Balancing policy (177.5s) is 22% faster than the Growth policy (226.6s) and the PID Controller policy has a gain of 8% (208.9s).

In order to highlight the differences between the PID Controller and the Balancing policy, which processed the workloads similarly fast, Figure 12 shows the number of re-

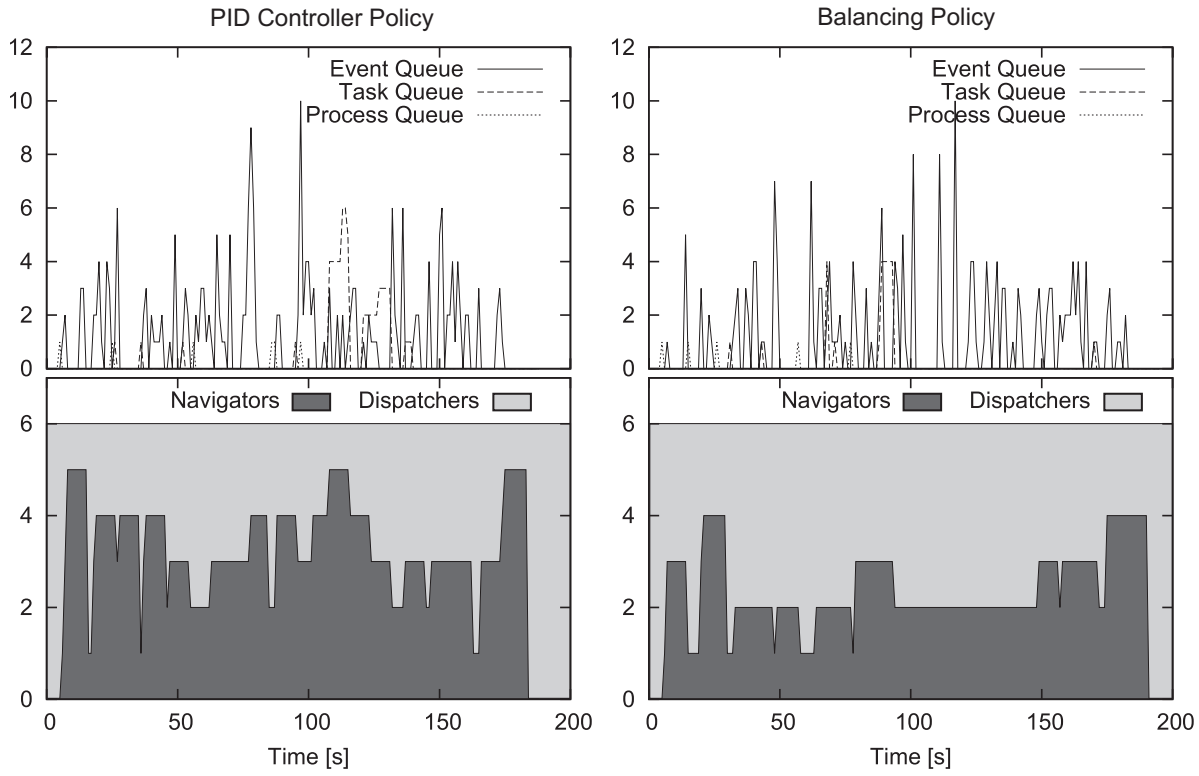

Figure 10. fMRI Workload Execution Plots

configurations required for the two policies. In case of all workloads, the Balancing policy performs fewer reconfigurations, indicating that in this case the configuration oscillates less during the course of an experiment.

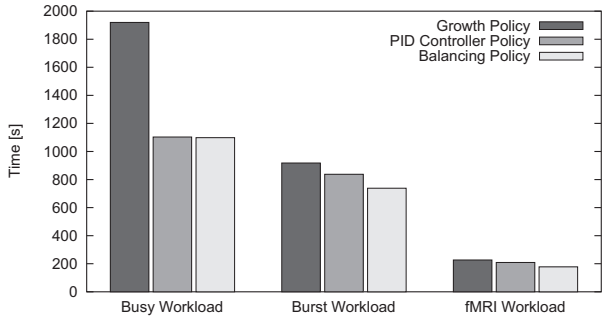

Figure 11. Performance Comparison

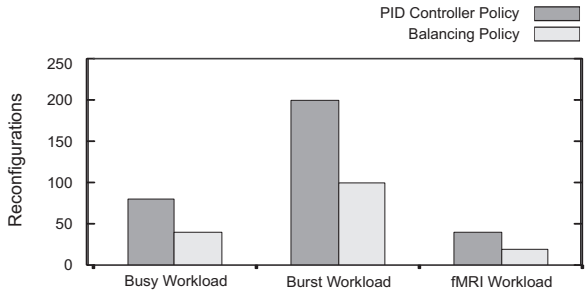

Figure 12. Number of Reconfigurations

7. Related Work

Zero-configuration is a very important requirement for improving the acceptance and usability of systems. This has been recognized, for example, in the area of networking for some time [4].

In the context of autonomic computing, the problem and the difficulty of configuring controllers based on low-level performance indicators has been recognized [3]. The use of higher-level policies and goals has also been studied [10]. Other approaches [13] express the goals in the form of quality of service parameters which need to be set to define the behavior of the system. Using these parameters, a QoS controller estimates the QoS provided by the system in the short-term, matches it with the desired QoS and performs configuration changes to optimize the system with respect

to the QoS goal. In our approach, we propose to employ policies that remove the need for any configuration of the controller.

A more holistic approach [11] goes beyond reconfiguring the engine and also targets at reconfiguring the complete environment: possible reconfiguration actions for instance also include moving services used by the workflow between nodes. Instead of reconfiguring the engine, also the workflow can be reconfigured in order to react to a changing environment [2]. Using late binding, the services used by the workflow are determined during execution through metadata attached to the workflow.

8. Conclusions

Many autonomic systems achieve self-configuration through a controller component, which monitors a system's operations as well as performance and reacts to imbalances due to workload changes by applying reconfiguration actions. Whereas employing such a controller can remove the need for manual system configuration, most simple controllers depend on thresholds and if-then rules, whose parameters still require tuning. In this paper we have shown that the performance of an autonomic system can be very sensitive to the configuration of its autonomic controller. We argued that manual configuration of an autonomic controller defeats the purpose of self-configuration and that zero-configuration control policies should be applied instead. We include an experimental study of two such policies, one based on a standardized PID controller, for which an extensive literature on tuning techniques is available. The second is based on an analytical model applicable to stage-based architectures, where the controller ensures that the rate of message production and consumption through a queue remains balanced. This policy can self-tune its operating parameters based on observable properties of the system and thus requires zero-configuration. Our evaluation in the context to the JOpera autonomic workflow engine has shown the feasibility of using zero-configuration policies for realistic workloads. Not only the proposed policies do not require any manual configuration, but they provide a significant performance gain over simpler policies based on thresholds, even when these are optimally tuned.

Acknowledgements

The authors would like to thank Gustavo Alonso and Michal Young for the many useful comments and Lukas Füllemann for helping with the experiments.

This work is partially supported by the EU-IST-FP7-215605 (RESERVOIR) project, the EU-IST-FP6-15964 (AEOLUS) project and by a grant from the Hasler Foundation (ManCom Project No. 2077).

References

[1] JOpera Website. http://jopera.org.

[2] R. Baird, M. Hepner, R. Gamble, and M. T. Gamble. Reconfiguring workflows of web services. In *ICCBSS '07: Proc. of the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, pages 131–140, 2007.

[3] D. Breitgand, E. Henis, and O. Shehory. Automated and adaptive threshold setting: Enabling technology for autonomy and self-management. In *ICAC '05: Proc. of the Second International Conference on Automatic Computing*, pages 204–215, Washington, DC, USA, 2005. IEEE Computer Society.

[4] S. Cheshire and D. Steinberg. Zero configuration networking: The definitive guide. December 2005.

[5] P. Cominos and N. Munro. PID controllers: recent tuning methods and design to specification. *Control Theory and Applications, IEE Proc.* -, 149(1):46–53, Jan 2002.

[6] T. Heinis, C. Pautasso, and G. Alonso. Design and evaluation of an autonomic workflow engine. In *ICAC '05: Proc. of the 2nd International Conference on Autonomic Computing*, pages 27–38, 2005.

[7] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[8] J. D. V. Horn. Online availability of fMRI results images. *Journal of Cognitive Neuroscience*, 15(6):769–770, 2003.

[9] J. Kephart. Research challenges of autonomic computing. In *ICSE '05: Proc. of the 27th international conference on Software engineering*, pages 15–22, 2005.

[10] J. Kephart and S. White. A research agenda for business-driven information technology. In *HotAC I: Hot Topics in Autonomic Computing*, Dublin, Ireland, 2005.

[11] K. Lee, R. Sakellariou, N. Paton, and A. Fernandes. Workflow adaptation as an autonomic computing problem. In *WORKS '07: Proc. of the 2nd Workshop on Workflows in Support of Large-scale Science*, pages 29–34, 2007.

[12] G. P. Liu and S. Daley. Optimal-tuning PID control for industrial systems. *Control Engineering Practice*, 9(11):1185–1194, November 2001.

[13] D. A. Menascé and M. N. Bennani. On the use of performance models to design self-managing computer systems. In *Proc. of Computer Measurement Group Conference*, pages 1–9, 2003.

[14] J. Parekh, G. Kaiser, P. Gross, and G. Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006.

[15] C. Pautasso, T. Heinis, and G. Alonso. Autonomic execution of web service compositions. *ICWS 2005: Proc. of the Third IEEE International Conference on Web Services*, July 2005.

[16] C. Pautasso, T. Heinis, and G. Alonso. Autonomic resource provisioning for software business processes. *Information and Software Technology*, 49(1):65–80, 2007.

[17] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[18] J. Ziegler and N. Nichols. Optimum settings for automatic controllers. *ASME Trans.*, 64:759–768, November 1942.