Self-Adapting Service Level in Java Enterprise Edition

Jérémy Philippe[1], Noël De Palma[1 2] Fabienne Boyer[1], and Olivier Gruber[3]

[1] INRIA Rhône-Alpes, France
[2] Grenoble Institute of Technology
[3] University of Grenoble I
Firstname.Lastname@inria.fr

Abstract. Application servers are subject to varying workloads, which suggests an autonomic management to maintain optimal performance. We propose to integrate in the component-based programming model often used in current application servers the concept of *service level adaptation*, allowing some components to dynamically degrade or upgrade their level of service. Our goal is to be able, under heavy workloads, to trade a lower service level of the most resource-intensive components for a stable performance of the server as a whole. Upgrading or degrading components is autonomously performed through runtime profiling, which is used to estimate the application's hot spots and target adaptations. In addition to finding the best adaptations, this performance profile allows our system to characterize the effects of past adaptations; in particular given the current workload, it is possible to estimate if a service level upgrade might result in an overload. As a result, by stabilizing the server at peak performance via component adaptations, we are able to drastically improve both overall latency and throughput. For instance, on both the RUBiS[4] and TPC-W benchmarks[5], we are able to maintain peak performance in heavy load scenarios, far exceeding the initial capacity of the system.

Keywords: Quality of service, service-level degradation, control loop, performance profile, self-adaptation.

1 Introduction

Autonomic management is increasingly important, especially regarding adaptive behaviors in the presence of varying workloads. Application servers openly available on the Internet are especially subject to such workloads and offer the incentive to design and evaluate adaptive behaviors. Some of our previous work has studied autonomic optimization exploiting load balancing in clusters [1]. This work focuses on exploiting application-level adaptations that are naturally present in Internet applications.

[4] http://rubis.objectweb.org/

[5] http://www.tpc.org/tpcw/

Indeed, it is our experience that components of Internet applications often contain the opportunity for behavioral adaptations. A common example of such adaptations can be found in the context of multimedia streaming servers, where the resolution and encoding of the content can be adapted to control the demand in CPU and network bandwidth [2]. Limiting the size and precision of search results is also a well-known and efficient adaptation of Internet applications. Sorting is always an expensive operation on large results, which may be avoided or approximated in some cases. Transactions are also a classical source of overheads that can be mitigated through smaller transactions, less consistency, or playing with the granularity of locks.

In our approach, we request component designers to explicit possible behavioral adaptations, in the form of alternative (and generally degraded) service levels. Each component can be individually moved up or down that sequence, raising or lowering the level of the provided service. At lower levels, a component generally uses less resources to provide its service. Explicit levels of service offer autonomic managers the opportunity to adapt the overall resource usage of an application, trading a lower service level of individual components for an improved quality of service of the application as a whole (i.e. better latency and peak throughput). We use a dynamic approach in which an autonomic manager decides to degrade or upgrade service levels at runtime, based on workload fluctuations.

The decision to apply or unapply an adaptation is fully autonomic. We only request that component designers express service levels. We felt important that we do not require them to measure or estimate the resource usage of these service levels. Indeed, such estimates are not only difficult to make accurately for an individual component but are almost impossible to make when considering all possible architectures and combinations of service levels for other components. To estimate resource usage, we rely on a traditional profiling technique based on request sampling, that we tailor to our component-oriented architecture. In particular, we abstract the traditional call stack into a more abstract *component stack* that provides an execution pattern in the sampled system. For each such execution pattern, we can estimate its intrinsic cost per resource. Using this intrinsic cost, we can calibrate the gains of adaptations per component stack and per resource. Through such gains, we learn about past effects of adaptations, helping us to optimize future adaptation decisions.

The challenge of this approach is to obtain *adaptation gains* that are workload independent. Indeed, gains are estimated on past workloads and used to predict effects of adaptation on future workloads. Using our knowledge of the architecture, we estimate our adaptation gains at the fine-grain level of component stacks, achieving enough workload independence. Our experiments show that effectively, our autonomic manager accurately estimates the effects of adaptations and efficiently corrects both overload and underload situations, even in the presence of varying workloads.

We prototyped our autonomic adaptation system in the context of Internet application servers based on the Java EE model (Java Enterprise Edition). This prototype is an extension of the open-source JOnAS middleware. The modifications are minimal and incur no significant overhead. In particular, our continuous 10Hz sampling incurs no measurable overhead in both RUBiS and TPC-W benchmarks. Our sampling rate is enough to compute component stack costs with good precision and thereby measure reliable adaptation gains. Our experiments show that our system consistently improves the overall performance of JEE applications under heavy workloads, both reducing latency and increasing throughput. Our experiments also show that our system is not prone to oscillations and adapts quickly to changing workloads.

The rest of this paper is organized as follows. In Section 2, we present the design of our autonomic adaptation system based on techniques from control theory. In Section 3, we details our sampling techniques and how we approach workload-independent gains. In Section 4, we present a simple example illustrating our performance metric and simple adaptive behaviors. In Section 5, we discuss the adaptive behavior obtained on the RUBiS and TPC-W benchmarks. In Section 6, we discuss related work. In Section 7, we conclude.

2 Autonomic Adaptation

Our autonomic adaptation system uses techniques from control theory, which has become a common practice in autonomic systems [3][4][5]. The configuration of our control loop is composed of two thresholds defined for each resource. The overload threshold is the usage ceiling above which the controller looks for a service degradation to lower the usage of the overloaded resource. The underload threshold is the usage floor upon which the controller *may* consider a service upgrade.

Our adaption system follows the simple state machine depicted in Figure 1. The adaptation system has a regulation mode and a calibration mode. In regulation mode, the control loop monitors the load of each resource. It reacts to overload situations by selecting the most efficient adaptation that is not yet applied and applies it. It reacts to underload situations by selecting amongst already applied adaptations which one is the most effective to unapply. After each regulation, the adaptation system steps into calibration mode for a fixed calibration period. During this calibration period, further regulations are inhibited.

In calibration mode, the autonomic adaptation system measures the impacts on resource usage of the adaptation it just applied. The calibration period has been experimentally fixed to 10 seconds, which is neither too short nor too long. Too short, we would not be able to accurately estimate the effects of a regulation on resource usage. Too long, changes in workloads could interfere with our estimate. Moreover, a long calibration delay hinders regulation since regulations are inhibited during calibration. Calibration will be detailed in Section 3.

Regarding regulation, one of the main challenges is stability, which we address using an asymmetrical selection of adaptations. In the overload case, the control loop looks for an adaptation δ to apply with a high effect on resource usage

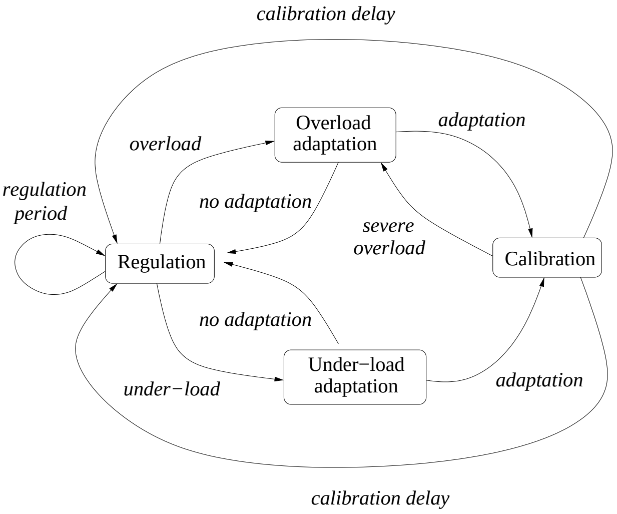


Fig. 1. Autonomic Adaptation Design

(noted $\Delta_\delta U_R$) and a low degradation on the level of service, noted w_δ. This w_δ is provided by component designers who annotate their adaptations with their relative impact on the quality of service of their component. The $\Delta_\delta U_R$ is automatically estimated by our system, which is detailled later on. This overload goal is captured by the following definition where $E(\delta^+)$ is the efficiency of applying the adaptation δ:

$$E(\delta^+) = \frac{\Delta_\delta U_R}{w_\delta}$$

In the underload case, the control loop looks for an adaptation to unapply with a low effect on resource usage and a high improvement of the level of service. This goal is captured by the following definition where $E(\delta^-)$ is the efficiency of unapplying the adaptation δ.

$$E(\delta^-) = \frac{w_\delta}{\Delta_\delta U_R}$$

It is important to point out that these definitions imply, through $\Delta_\delta U_R$, a mean to estimate the effects of adaptation δ on resource usage. Also, estimating impact on resource usage is important to make sure that an adaptation targeting a particular resource will not overload another resource (in the case of adaptations intended to find a trade-off between several resources). To estimate $\Delta_\delta U_R$, we define characteristics called *adaptation gains*, which are evaluated based on the observed effects of the past regulations that used adaptation δ.

3 Adaptation Gains

The gain of an adaptation captures the effects on resource usage of that adaptation. We estimate the gain of an adaptation when it is applied under a certain workload but we want to predict the effects of applying or unapplying that same adaptation at some later time under a potentially unrelated workload.

The challenge is therefore to characterize the gain in a way that is as much workload independent as possible. We compute the gain when we calibrate by measuring the usage delta of a resource R which results from applying adaptation δ:

$$\Delta_\delta U_R = U_R^+ - U_R^-$$

U_R^+ is the usage of resource R sampled and averaged during the calibration delay, after adaptation δ is applied. U_R^- is the usage of resource R sampled and averaged right before adaptation δ is applied.

A simple approach to modeling the gain of adaptation δ on resource R could be to define the gain $G_\delta(R)$ as follows:

$$G_\delta(R) = \frac{U_R^+}{U_R^-}$$

However, this simple approach does not adequately isolate the effects of the adaptation δ. This gain captures the usage delta of the resource R due not only to tasks executing within the adapted component but also due to tasks whose executions are never touching the adapted component. Changes of workload, unrelated to the adaptated component, happening during calibration, could affect our gain estimate. In particular, the more the adapted component is involved in the workload, the higher the impact on resource usage. A better approach is to focus our gain estimation solely on tasks whose executions involve the adapted component.

We therefore need to separately account resource usage depending on the components involved in the tasks, which we achieve through a profiling technique called statistical sampling [6][7]. The traditional approach periodically captures the call stacks of active threads in a system. In our approach, we extract *component stacks* from call stacks as depicted in figure 2. In this example, we have a simple assembly of components in the architecture: a component A connected to two components B and C. We show the call stack of one active task, making function calls in component A and B. The corresponding component stack, noted $A - B$, abstracts away from the individual stack frames, providing an execution pattern (or signature) for the currently executing tasks from an architectural point of view.

Once we have component stacks, we can link them to resources as follows. Typically, we start by modelling the processor as a *CPU resource* and I/O subsystems as *I/O resources*. In a sample, we relate each component stack appearing

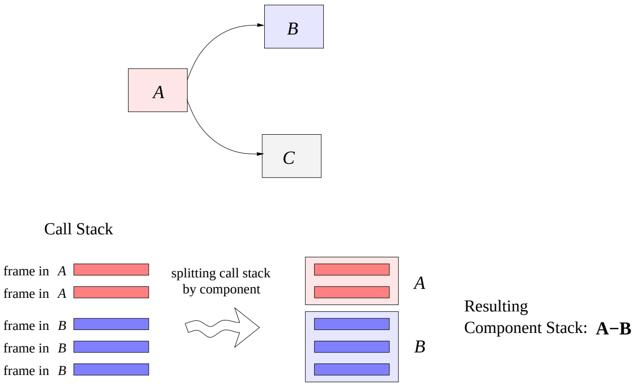


Fig. 2. From call stacks to component stacks

in that sample with one and only one resource. Given the component stack of an active task, we associate that component stack with an I/O resource R if the corresponding task is explictely waiting for I/O on that resource R when sampled. Otherwise, the component stack is associated with the CPU resource.

Once component stacks are associated with resources, we define for each sample the *hit rate* of a component stack. The hit rate of a component stack S, noted $H(S)$, is the percentage of component stacks S in the sample that are associated with the same resource R. Once we have the hit rate of component stacks, we compute for each sample the usage of the associated resource R by a given component stack S, which defined as follows:

$$U_R(S) = H(S).U_R$$

U_R is the overall usage of the resource R given by the operating system at the time of the sample. Having the resource usage per component stacks offers a better gain estimate, defined as follows:

$$G_R^\delta(S) = \frac{U_R^+(S)}{U_R^-(S)}$$

In practice, we experienced too much volatility when considering a single sample. In our empirical tests, we have found that averaging the samples over a fixed window (simple moving average) yields good results. Even using averaged $U_R(S)$, the previous gain is not workload independent enough. We certainly made progress since we focus our estimation on gains per component stacks on a resource R. However, this gain is still sensitive to the actual execution count of each component stack, which may vary from one workload to another.

To illustrate this variability, we assume that we have an adaptation δ that improves by 50% the CPU usage. With a constant workload, applying this adaptation would produce an estimated gain close to 0.5 for all stacks using the adapted component. Other stacks would see a gain close to one. However, a varying workload would affect these gain estimates. Indeed, if we assume that we have many queued requests when we regulate (a common case in an overload situation), the idle CPU time that our regulation just freed is likely to be used to process some of the pending requests, right during our calibration. If these processed requests trigger the pattern S, $H(S)$ will increase since we have more executions of the stack S. This may yield a $G_R^\delta(CPU)$ that could be much higher than 0.5, i.e. the adaptation appears less efficient than it actually is. To remove this variability, we need to measure the execution rate of component stacks and introduce the *cost* of a stack S on a resource R, noted $C_R(S)$ and defined as follows:

$$C_R(S) = \frac{U_R(S)}{W(S)}$$

$W(S)$ captures the execution rate (or workload) of the component stack S, computed during each sample. $W(S)$ is obtained by counting the number of times a task enters a component with stack S. Using the cost rather than the resource usage of stacks, we can measure a gain that remains fine grain at the level of a single execution pattern and that becomes fairly independent of workload changes. Thus, the final definition of our gain is as follows:

$$G_R^\delta(S) = \frac{C_R^+(S)}{C_R^-(S)}$$

$C_R^+(S)$ is the cost of the stack S estimated right after we apply the adaptation δ. $C_R^-(S)$ is the cost of the stack S right before we apply the adaptation δ.

Using the measured gains $G_R^\delta(S)$ of adaptations that we applied in the past, we can make a good estimation of the future effects of these adaptations even if the workload changes. Through runtime sampling, we know the complete performance profile of the managed system: all resource usages $(\forall R, U_R)$, the active stacks, their hit rate $(H(S))$, and their costs per resource $(C_R(S))$. Figure 3 summarizes these metrics and their meaning.

Using the current performance profile, our control loop can estimate the variation $(\Delta_\delta U_R)$ on resource usage of applying or unapplying a given adaptation δ as the sum over all stacks in the current profile of the effects on the usage of the resource R:

$$\Delta_\delta U_R = \sum_{\forall S} (U_R^+(S) - U_R^-(S))$$

Since we have:

Metric	*Name*	*Definition*	*Unit*
$H(S)$	Hit ratio	Proportion of tasks with stack S in the profiling samples	None
$U_R(S)$	Resource usage	Proportion of time the resource R is used by a task with stack S	None
$W(S)$	Execution rate	Frequency at which tasks call the stack S	Hertz
$C_R(S)$	Cost	Usage duration of resource R each time the stack S is called by a task	Seconds

Fig. 3. Metrics used to define and measure adaptation gains

$$C_R(S) = \frac{U_R(S)}{W(S)}$$

We then have:

$$\Delta_\delta U_R = \sum_{\forall S} (C_R^+(S).W(S) - C_R^-(S).W(S))$$

We can express in this formula $C_R^+(S)$ with both $C_R^-(S)$ and $G_R^\delta(S)$. However, we have to consider the overload and underload case separately. In the case of an overloaded resource R, we expresss $C_R^+(S)$ as follows:

$$C_R^+(S) = G_R^\delta(S).C_R^-(S)$$

We therefore have:

$$\Delta_\delta U_R = \sum_{\forall S} (G_R^\delta(S) - 1).C_R^-(S).W(S) \tag{1}$$

In the case of underloaded resource R, we estimate $C_R^+(S)$ differently from $C_R^-(S)$ and $G_R^\delta(S)$:

$$C_R^+(S) = \frac{C_R^-(S)}{G_R^\delta(S)}$$

We therefore have:

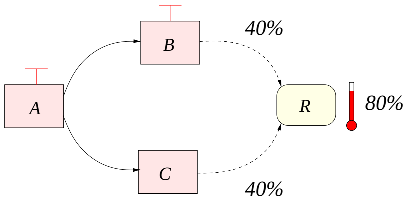
$$\Delta_\delta U_R = \sum_{\forall S} \frac{(1 - G_R^\delta(S))}{G_\delta(S)}.C_R^-(S).W(S) \tag{2}$$

Using formula (1) or (2), our system can estimate accurately the effects of applying or unapplying the adaptation δ in the current workload. The estimate is accurate because the metrics are obtained at the fine granularity of individual stacks and we only sum the estimated effects for the relevant stacks. The relevant stacks are the very stacks identified in the last performance profile, which characterizes the current workload.

4 Example

Figure 4 represents a simple architecture with a resource R and three components A, B, and C. The table represents a possible performance profile for this system, showing our metrics $H(S)$, $U_R(S)$, $W(S)$ and $C_R(S)$. In this performance profile, the usage of resource R, $U_R(S)$, is 80%. The performance profile also shows the active component stacks associated with R: stack $A - B$ and stack $A - C$.

A performance profile allows to observe *how* a system uses its resources. For instance, the resource usage metric $U_R(S)$ shows that stack $A - B$ causes the same usage of R as stack $A - C$. Furthermore, the execution rate and cost $W(S)$ and $C_R(S)$ show that the cost of $A - B$ is lower than the cost of $A - C$ since both stacks cause equal resource usage, while $A - B$ receives an higher workload.

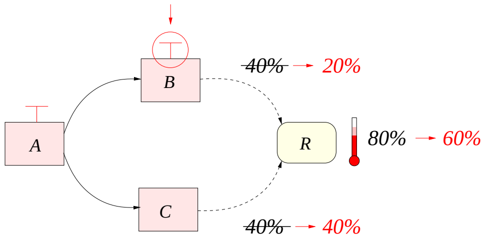


S	$U_R(S)$	$W(S)$	$C_R(S)$
A-B	40%	10 Hz	40 ms
A-C	40%	5 Hz	80 ms

Fig. 4. Example of a performance profile

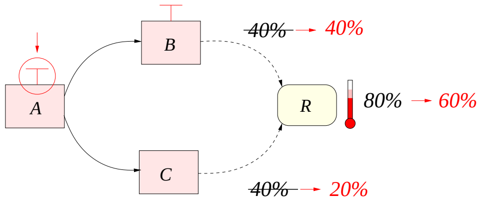
Based on this performance profile, we now illustrate the results of several possible adaptations. Suppose that if we degrade component B, the degradation produces the effects described in the performance profile shown in figure 5. We see that the service degradation mechanism provided by B lowers the cost of stack $A - B$ from 40ms down to 20ms, but has no effect on the cost of stack $A - C$. This illustrates that adapting a component usually does not impact the stacks that are not involved with the adapted component.

Instead of degrading B, suppose that if we degrade the service level of A, this produces the effects described in figure 6. We see that the service degradation mechanism provided by A lowers the cost of stack $A - C$ from 80ms down to 40ms, but has no effect on the cost of stack $A - B$. This illustrates that adapting a component does not always impact all component stacks equally, i.e. an adaptation can affect only some of the tasks involving the adapted component.



S	$U_R(S)$	$W(S)$	$C_R(S)$	$G_R^{\delta A}(S)$	$G_R^{\delta B}(S)$
$A-B$	~~40%~~ → 20%	10 Hz	~~40 ms~~ → 20 ms	??	~~??~~ → 0.5
$A-C$	~~40%~~ → 40%	5 Hz	80 ms	??	~~??~~ → 1.0

Fig. 5. Adapting component B, constant workload



S	$U_R(S)$	$W(S)$	$C_R(S)$	$G_R^{\delta A}(S)$	$G_R^{\delta B}(S)$
$A-B$	~~40%~~ → 40%	10 Hz	40 ms	~~??~~ → 1.0	0.5
$A-C$	~~40%~~ → 20%	5 Hz	~~80 ms~~ → 40 ms	~~??~~ → 0.5	1.0

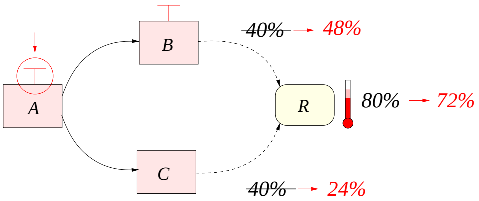
Fig. 6. Adapting component A, constant workload

Notice that we have no variation of workload in the above example. However, consider the case shown in figure 7, which depicts the effects of the same adaptation on component A but as the execution rate increases because of request queueing. We see that if our gain estimates were based on resource usage only, the adaptation gains would be incorrectly estimated. By using the cost metric, these estimations are protected from workload fluctuations.

5 Evaluation

5.1 Implementation requirements

We have prototyped our autonomic adaptation system in the context of multi-tier Java EE application servers, which are based around a presentation tier (Servlets/JSP), a business logic tier (Enterprise Java Beans – EJBs) and a database tier. To enable adaptation capabilities, we have slightly extended the Java EE model to allow EJB components to provide alternative runtime modes. These alternative modes correspond to degraded or improved service levels, which can be enabled or disabled dynamically, either by the application server



S	$U_R(S)$	$W(S)$	$C_R(S)$	$G_R^{\delta A}(S)$	$G_R^{\delta B}(S)$
$A-B$	~~40%~~ → 48%	~~10 Hz~~ → 12 Hz	40 ms	1.0	0.5
$A-C$	~~40%~~ → 24%	~~5 Hz~~ → 6 Hz	~~80 ms~~ → 40 ms	0.5	1.0

Fig. 7. Adapting component A, changing workload

(programmatically) or by a human administrator (interactively). This dynamic configuration is currently done through the JMX API (Java Management Extensions).

Otherwise, we rely on the standard Java EE concepts of components and dependencies: capturing Servlets, EJBs and databases as components with explicit dependencies in the architecture. However, recall that our approach also requires that physical resources are made explicit in the model, to be able to link component stacks with resources. We currently do this at a relatively coarse grain through the application server's knowledge of the physical machine(s) used by each tier. For instance, a component stack that corresponds to an EJB component will be associated to the machine (or set of machines) running the EJB tier.

Then, to profile the application, we extend the application server in order to capture component stacks and count their execution rates. To achieve this, we associate each request with a *profiling context* that contains its current stack and we intercept component calls to update the stacks and execution counts. To make sure that the stack has a global scope, this context must be propagated both through local component calls (using a thread-local variable) and through remote component calls (using serialization mechanisms). Most often, such context propagation facilities are already present in Java EE application servers, for security and transaction management.

Once this low-level instrumentation is provided, our autonomic management extension is essentially composed of two services. A *profiling service* periodically samples the component stacks of all ongoing requests to produce the metric $H(S)$, and monitors their execution rates to produce the metric $W(S)$. Then, by monitoring resource usage, it produces the per-stack resource usage and cost metrics, $U_R(S)$ and $C_R(S)$. Secondly, an *adaptation service* implements our autonomic manager, by dynamically reacting to overload and underload conditions, using the profiling service to select optimal adaptations and estimate adaptation gains.

5.2 Software environment

Our prototype is an extension of the JOnAS application server. Software versions are as follows: Java v1.5, JOnAS v4.8, MySQL v5.0 and Fedora Core 6 Linux. Our test machines have the following specifications: Intel Core Duo 1.66 GHz, 2 GB memory, Gigabit Ethernet network. In our experiments, three machines are dedicated to the application (one machine per tier) and one machine is dedicated to load injection (except when running the two benchmarks together, in which case two machines are used to isolate the load injectors).

Our performance evaluation is based on the RUBiS and TPC-W benchmarks. RUBiS simulates an online auction application [8]. Load injection in RUBiS is configured by a transition matrix, and two specific matrices are generally used to produce either a read-only workload or a read-write workload. Regarding TPC-W, we have used the implementation from Rice University, which is based on Servlets only. Since our prototype is based on adaptable EJB components, we have modified this implementation, wrapping the JDBC calls with session beans. An interesting side-effect of this modification is to produce a finer-grained component-oriented description. Load injection in TPC-W is also configured by transition matrices. The TPC-W specification defines a read-only matrix (browsing mix), a write-20% matrix (shopping mix), and a write-50% matrix (ordering mix).

5.3 Profiling overhead

Our first experiment shows that the overhead of our profiling mechanism is negligible in the context of these benchmarks. We begin by noting that profiling overhead is mostly dependent on the following two factors:

- Interception of component calls, proportional to throughput.
- Request sampling, proportional to sampling frequency.

To measure the profiling overhead, we first checked that there is no performance difference between the baseline system (running the benchmarks in an unmodified environment) and the instrumented system when profiling is used with a very low sampling frequency (e.g. 0.1 Hz). Then, we measured the benchmark's peak performance for increasing sampling frequencies, since this parameter is crucial in controlling both the profiling precision and its overhead. As figure 8 shows in the case of the RUBiS benchmark, we have not been able to detect a significant overhead, even for high sampling frequencies. In practice, we observed that a frequency as low as 1 Hz provides a reasonable precision for the purpose of adaptation (although the following experiments were done with a 10 Hz frequency to improve precision). As a side knowledge, this figure also shows that the CPU of the database tier is the bottleneck in RUBiS. TPC-W yields the same results as RUBiS (i.e. no overhead and the database is the benchmark's bottleneck).

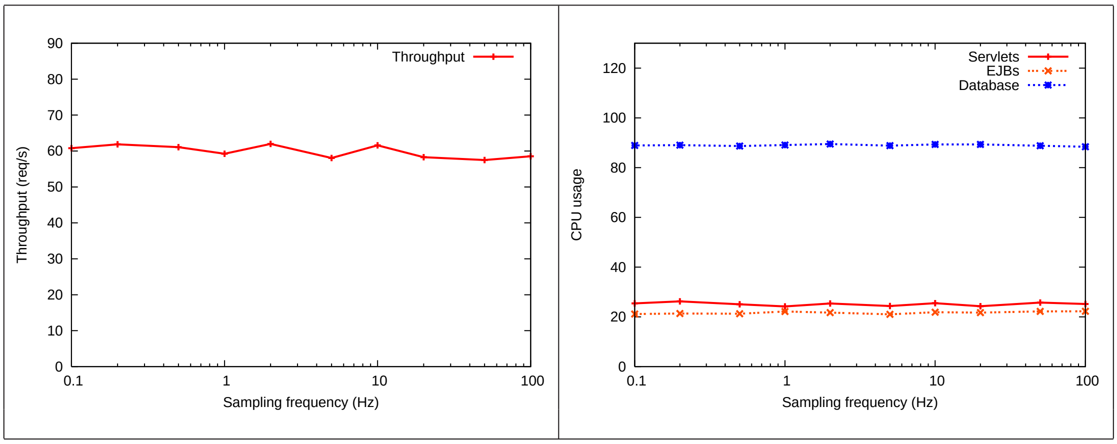


Fig. 8. RUBiS Performance vs. Sampling Frequency

5.4 RUBiS benchmark

In all experiments, calibration delay is set to 10 seconds and sampling frequency is set to 10 Hz. We first present the results of profiling RUBiS for a typical stationary workload: 256 emulated clients and a read-write mix, with 900 seconds of runtime. All metrics (hit rates, resource usage, workload and cost) were averaged over the entire experiment. We only show the performance profiles for the database CPU, since we already established that this resource is the bottleneck and the only one triggering adaptations. The database is about 180 MB for a little over forty thousands items and one hundred thousands customer records. The four most database intensive stacks (i.e. those with the highest hit rate) were:

Stack	H(S)	W(S)	C(S)
`SearchByCategory.Category`	64.1 %	6.60 Hz	55.8 ms
`SearchByRegion.Category`	30.4 %	2.22 Hz	78.4 ms
`AboutMe.User`	0.95 %	11.4 Hz	0.49 ms
`SearchByRegion.Item`	0.83 %	16.7 Hz	0.29 ms

These results show that only two component stacks are responsible for most of the CPU usage on the database tier. These stacks—`SearchByCategory.Category` and `SearchByRegion.Category`, are associated to the search for auctioned items. Consequently, we have implemented two adaptations, both based on deactivating sorting—a costly operation on the database side, especially with large tables. One adaptation is on the `SearchByCategory` component and the other is on the `SearchByRegion` component. Our results show that the adaptation on `SearchByCategory.Category` is significantly more effective than the one on `SearchByRegion.Category`. This is because the former generally involves sorting more items than the later; the straightforward consequence of a less selective filter.

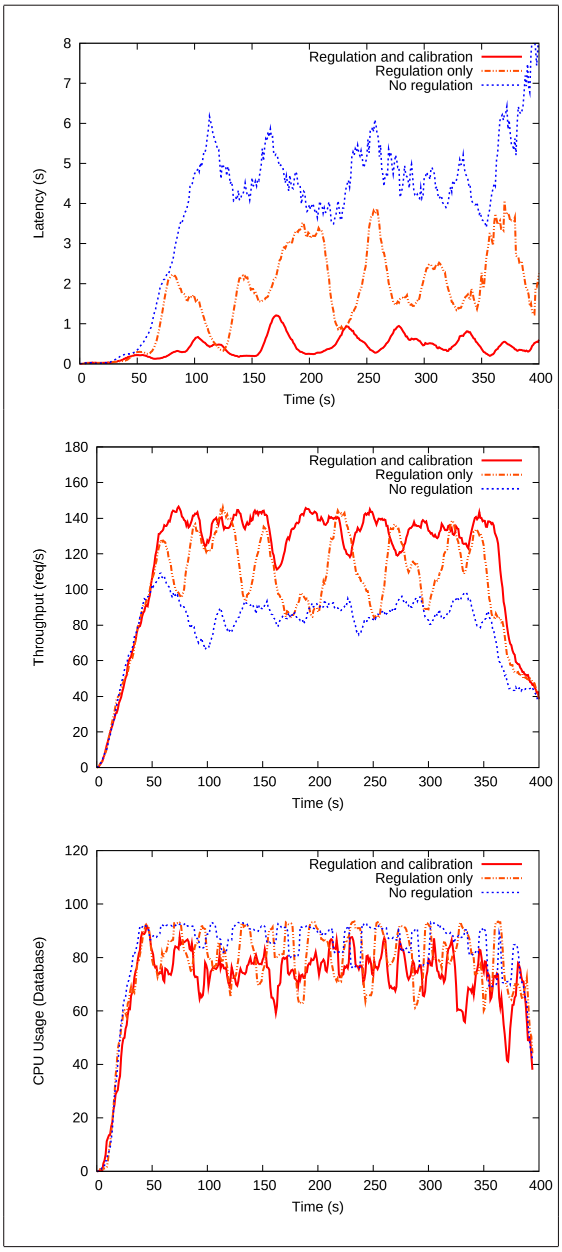


Fig. 9. RUBiS Performance

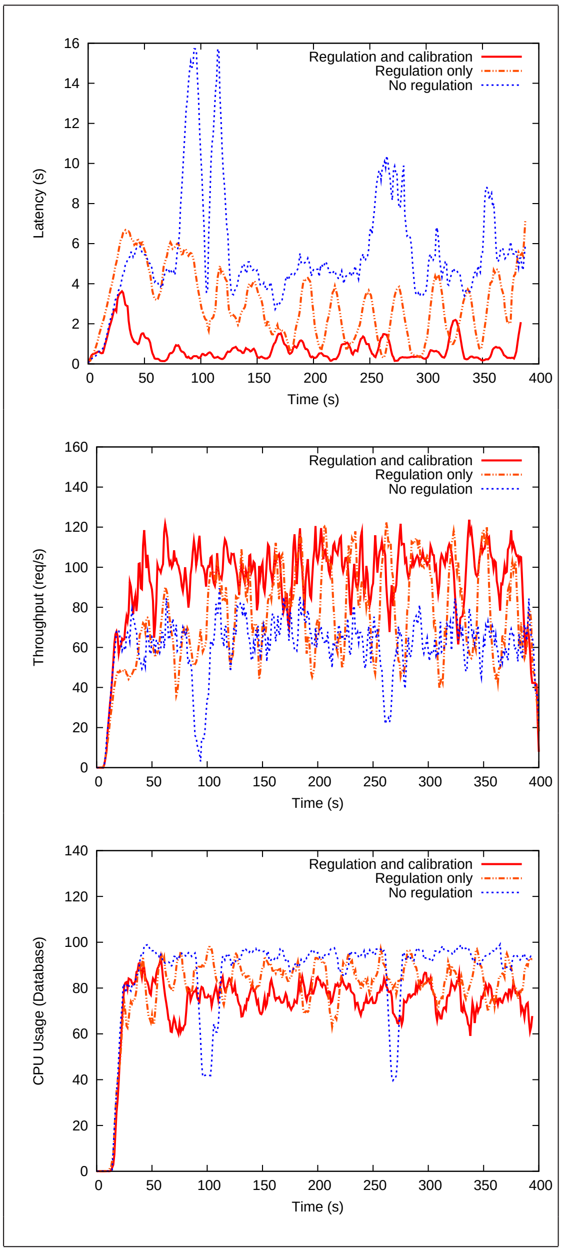


Fig. 10. TPC-W Performance

To evaluate our dynamic adaptation, we configure a workload that far exceeds the normal benchmark capacity (about 650 clients): 1024 emulated clients and a read-write mix, with 60 seconds of ramping up and 300 seconds of runtime.

We are targeting an agressive goal in terms of CPU usage: overload threshold of 90% and underload threshold of 80%. The rationale is to show that we can target a high and narrow window of CPU usage, one that is sufficiently high for reaching and staying at peak performance but consistently avoiding the trap of thrashing. Only an automatic approach, with an accurate prediction, can attempt this; most systems have to be more conservative regarding their CPU usage target.

Figure 9 shows our results for RUBiS. We can see that our system maintains the database CPU around 80%, dividing latency by ten and improving throughput in the order of 75%. We can see that without regulations, the database CPU is consistently thrashing at about 95%, which explains the large latency and the poor throughput. With regulations but without calibration, we observe harmful oscillations since our system cancels adaptations as soon as the resource usage crosses the underload threshold. These oscillations are especially visible on latency and throughput where the performance with regulation but without calibration oscillates between the performance without regulation at all and the performance with regulation and calibration.

5.5 TPC-W benchmark

We present the results of profiling TPC-W for stationary workload of 256 clients and a shopping mix. Like RUBiS, our experiments show that the bottleneck is the database CPU. The database is about 1.1GB, with an extra 2.6GB for images (stored as static files). The database contains about 28.8 millions clients and 10,000 books. The four most sampled stacks were:

Stack	H(S)	W(S)	C(S)
`execute_search.author_search`	39.0 %	1.98 Hz	119.2 ms
`best_sellers`	28 % %	1.73 Hz	97.5 ms
`execute_search.title_search`	18.4 %	2.03 Hz	54.9 ms
`buy_confirm`	2.5 %	2.25 Hz	6.75 ms

These results show that three component stacks are almost equally responsible for most of the CPU usage on the database tier. These stacks—`execute_search.author_search`, `best_sellers`, and `execute_search.title_search`—are associated to the search for books, by authors, by title, or by best sellers. Consequently, we have implemented three adaptations, one for each stack. For the stacks `execute_search.author_search` and `execute_search. title_search`, the adaptation limits searching by looking for an exact match on titles or authors, avoiding costly substring matching. For the stack `best_sellers`, the adaptation looks for recent sellers as opposed to best sellers.

Like for our RUBiS experiments, we evaluate dynamic adaptation with a workload that exceeds the normal benchmark capacity (about 500 clients): 768 emulated clients and a shopping mix, with 60 seconds of ramping up and 300 seconds of runtime. We fixed the same agressive goal in terms of CPU usage, for the same reasons. Figure 10 shows our results for TPC-W. We can see that without regulations, the database tier is consistently thrashing with a CPU at about 95%, which explains the large latency and the poor throughput. We can also notice two sharp drop in CPU usage at time 100 and 260 seconds, that are totally avoided with our adaptive approach.

The CPU usage patterns and improvements are entirely consistent across the two benchmarks. With regulations but without calibration, our system is unable to predict the effects of applying or unapplying adaptations, which produces harmful oscillations. These oscillations are again quite visible on latency and throughput. With calibration, our system maintains the database CPU around 80%, dividing latency by ten and improving throughput in the order of 60%. Moreover, our system maintains a much more stable level of quality of service. Notice how much smoother the regulated latency and throughput are compared to the unregulated ones at 100, 250, and 350 seconds in the experiments. This is also visible in the much more stable CPU usage when regulated.

5.6 Combining TPC-W and RUBiS

To evaluate our system under non-stationary workloads, we combined both benchmarks as follows. We started TPC-W first and we started RUBiS about 200 seconds later. The two benchmarks therefore overlaps for about 200 seconds. During the final period of 300 seconds, we only have RUBiS. The latency and throughput results are depicted in Figure 11.

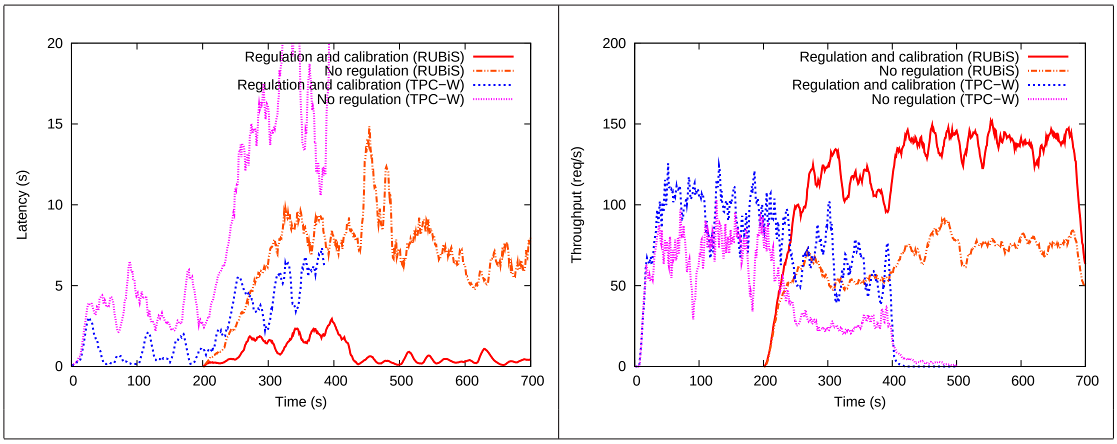


Fig. 11. TPC-W and RUBiS Combined

As expected, the first period shows very similar results to TPC-W alone. During the overlap period, the system is heavily overloaded. Without regulation, latency increases sharply and throughput drops significantly, for both benchmarks. With regulation, our system preserves as much overall quality of service as possible—it divides latency by 4 and almost doubles throughput. Furthermore, it is important to notice that our autonomic management reacts quickly to workload changes. Indeed, there is no visible period of instability as the workload changes, that is, when RUBiS starts at 200 seconds in the experiement and when TPC-W stops at 400 seconds in the experiment.

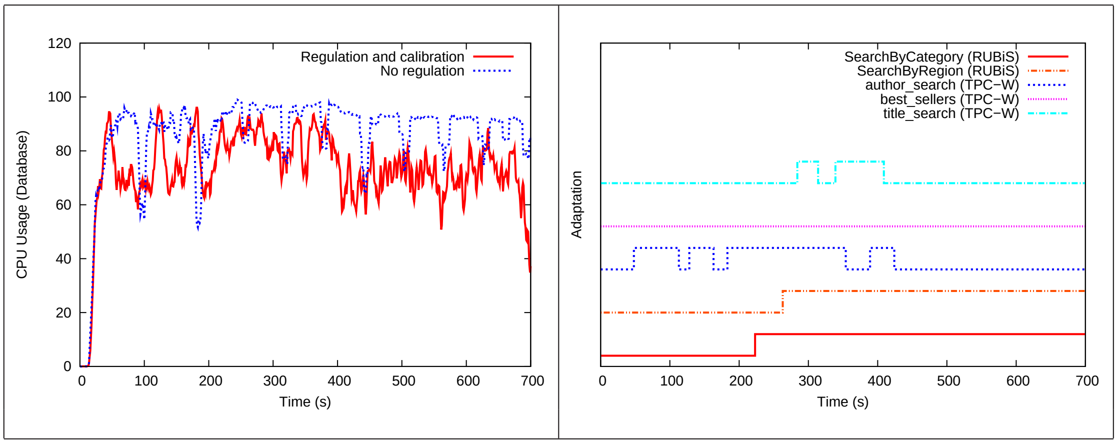


Fig. 12. CPU Usage and Adaptation Details

Figure 12 shows when adaptions are applied and unapplied during this experiment, combined with the CPU usage of the database tier. Up to about 300 seconds in the experiment, all available adaptations but one are dynamically and incrementally applied. Notice some oscillations happen. These oscillations are however few and sparse since we only have four of them in total, over 700 seconds, even though we consider applying or unapplying adaptations every 5 seconds. This means our prediction works, avoiding the vast majority of oscillations. This is confirmed by the fact that, although the CPU usage is often just below our 80% threshold, our autonomic manager maintains the adaptations, accurately predicting the CPU overload situation that would result if any adaptation would be unapplied.

Our approach mostly avoids oscillations, but large variations in resource usage may temporarily trick our prediction. Notice that no oscillation happens during the last period where RUBiS runs alone; this is because RUBiS has a more stable workload than TPC-W. Indeed, we noticed throughout our experiments a relatively higher instability in CPU usage for TPC-W, which we explain by the fact that TPC-W queries are more complex on a larger working set. While temporary low CPU usage may trigger a mistake, such mistakes are corrected quite rapidly. Furthermore, we argue that such mistakes induce an acceptable volatility in latency and throughput, most of the time within 20%. In particular, notice that this volatility never resulted in our experiments in the regulated quality of service dropping below the unregulated quality of service.

6 Related work

6.1 Service-level adaptation

Service-level adaptation provides an efficient mechanism to regulate resource consumption and avoid overload. However, it has been reserved in the past to specific types of systems, where adaptations are well-known and can be characterized in advance. For example, service adaptation has been used in the context of multimedia streaming servers, where increasing compression will save network bandwidth at the expense of content quality [2]. Another example resides in the context of security systems, where simpler encryption algorithms might require less processing while being less secure [9]. Similar examples can be found in the context of static web servers or distributed monitoring [10][11][12].

Our work contributes to service adaptation in the context of general distributed systems, where adaptations often cannot be characterized in advance. The key point of our approach is the dynamic construction of a performance profile, based on a component-based representation of the system.

6.2 Performance profiles

Performance profiles are used to locate and analyze the bottlenecks of a computing system [13]. A typical method to build performance profiles consists in tracking execution to find the code involved in each task, combined with resource usage measurements to find the most resource-intensive tasks.

Complex distributed systems make it difficult to use OS-level techniques because not only tasks often involve non-obvious sub-tasks but resources are also accessed through intermediate abstraction layers. One solution is to introduce new OS abstractions, but this is a complex and non-generic solution [14][15].

At the other end of the spectrum, another approach is to use statistical regression techniques to measure correlation between workload and resource usage [16][17]. This requires no instrumentation but only works if workload is highly variable (non-stationary). Furthermore, the system must be observed during a significant period to compute the regression with reasonable accuracy, which makes it inapplicable in the context of our work since our adaptation model requires the ability to observe the immediate effects of applied adaptations.

Our approach is an intermediate solution, based on the work of Chanda *et al.* [18][19]. First, a context is attached to each task and is propagated through the distributed system. This context contains the current path of the task through the distributed system. Then, statistical sampling is used to indirectly measure resource usage, and requires no intrusive OS-level instrumentation [20].

However, the contributions of Chanda *et al.* stop at computing performance profiles—using profiles for optimization purposes is left to a developer or an administrator. Our approach goes farther by considering an autonomic approach that leverages the knowledge of the component architecture of the observed system. Using component stacks, we can estimate gains and make useful predictions of component-level adaptations that can be used by a closed-loop control [21][22].

7 Conclusion

In this article, we have presented a novel approach to automatically regulate resource consumption in Internet application servers, such as Java EE servers, which often use a component-based programming model. Our proposition is to integrate the concept of service level adaptation to allow for automatically lowering the service level of individual components in order to preserve the overall performance in high-workload situations. By focusing adaptations on costly execution patterns, our approach optimizes service level while ensuring that resource usage does not exceed a predefined threshold.

The heart of our approach is a performance profile, which is used to estimate the effects of component adaptations. The challenge was to characterize these effects in a workload independent way so that the observation of past adaptation attempts could be reused to predict the effects of future adaptations, even if the workload is completely different. Combining runtime sampling and the knowledge of the component architecture, we designed the concept of adaptation gains at the granularity of component stacks. Our experiments show that our gains are workload independent enough so that our predictions are accurate and support our decision making to apply or unapply adaptations. The potentially harmful phenomenon of oscillations is kept to a minimum and results in no substantial instability in latency or throughput. Also, even when oscillations occur, the performance of the regulated system is always much higher than that of the baseline system.

References

1. Taton, C., Palma, N.D., Hagimont, D., Bouchenak, S., Philippe, J.: Self-Optimization of Clustered Message-Oriented Middleware. In: The 9th International Symposium on Distributed Objects, Middleware, and Applications (DOA), Vilamoura, Portugal (November 2007)
2. Layaida, O., Hagimont, D.: Designing Self-adaptive Multimedia Applications Through Hierarchical Reconfiguration. In: Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Athens, Greece (June 2005)
3. Diao, Y., Neha, G., Hellerstein, J.L., Parekh, S., Tilbury, D.M.: Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server. In: Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS), Florence, Italy (April 2002)
4. Abdelzaher, T.F., Shin, K.G., Bhatti, N.: Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. IEEE Transactions on Parallel and Distributed Systems (TPDS) **13**(1) (January 2002) 80–96
5. Kihl, M., Robertsson, A., Andersson, M., Wittenmark, B.: Control-Theoretic Analysis of Admission Control Mechanisms for Web Server Systems. World Wide Web Journal **11**(1) (March 2008) 93–116
6. Graham, S.L., Kessler, P.B., Mckusick, M.K.: gprof: a Call Graph Execution Profiler. ACM SIGPLAN Notices **17**(6) (June 1982) 120–126

7. Liang, S., Viswanathan, D.: Comprenhensive Profiling Support in the Java Virtual Machine. In: Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems (COOTS), San Diego, California, USA (May 1999)
8. Cecchet, E., Marguerite, J., Zwaenepoel, W.: Performance and Scalability of EJB Applications. In: Proceedings of the Symposium on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Seattle, Washington, USA (November 2002)
9. Wright, C.P., Martino, M.C., Zadok, E.: NCryptfs: A Secure and Convenient Cryptographic File System. In: Proceedings of the USENIX Annual Technical Conference, San Antonio, Texas, USA (June 2003)
10. Abdelzaher, T.F., Bhatti, N.: Web Content Adaptation to Improve Server Overload Behavior. In: Proceedings of the World Wide Web Conference (WWW), Toronto, Canada (May 1999)
11. Elnozahy, M., Kistler, M., Rajamony, R.: Energy Conservation Policies for Web Servers. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS), Seattle, Washington, USA (March 2003)
12. Sadler, C.M., Martonosi, M.: Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks. In: Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys), Boulder, Colorado, USA (October 2006)
13. Menascé, D.A., Dowdy, L.W., Almeida, V.A.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall (2004)
14. Banga, G., Druschel, P., Mogul, J.C.: Resource Containers: A New Facility for Resource Management in Server Systems. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA (February 1999)
15. Blanquer, J., Bruno, J., Gabber, E., McShea, M., Özden, B., Silberschatz, A., Singh, A.: Resource Management for QoS in Eclipse/BSD. In: Proceedings of the FreeBSD Conference, Berkeley, California, USA (October 1999)
16. Stewart, C., Kelly, T., Zhang, A.: Exploiting Nonstationarity for Performance Prediction. In: Proceedings of the EuroSys Conference, Lisbon, Portugal (March 2007)
17. Zhang, Q., Cherkasova, L., Mathews, G., Greene, W., Smirni, E.: R-Capriccio: A Capacity Planning and Anomaly Detection Tool for Enterprise Services with Live Workloads. In: Proceedings of the ACM/IFIP/USENIX Middleware Conference, Newport Beach, California, USA (November 2007)
18. Chanda, A., Elmeleegy, K., Cox, A.L., Zwaenepoel, W.: Causeway: Operating System Support for Controlling and Analyzing the Execution of Multi-tier Applications. In: Proceedings of the ACM/IFIP/USENIX Middleware Conference, Grenoble, France (November 2005)
19. Chanda, A., Cox, A.L., Zwaenepoel, W.: Whodunit: Transactional Profiling for Multi-Tier Applications. In: Proceedings of the EuroSys Conference, Lisbon, Portugal (March 2007)
20. Froyd, N., Mellor-Crummey, J., Fowler, R.: Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In: Proceedings of the ACM International Conference on Supercomputing, Cambridge, Massachusetts, USA (June 2005)
21. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Transactions on Computers **36**(1) (January 2003) 41–50
22. Sicard, S., Boyer, F., de Palma, N.: Using Components for Architecture-Based Management: The Self-Repair Case. In: Proceedings of the International Conference on Software Engineering (ICSE), Leipzig, Germany (May 2008)