

Control-theoretical CPU allocation: Design and Implementation with Feedback Control

Martina Maggio^{1,2}, Henry Hoffmann²,
Anant Agarwal² and Alberto Leva¹

¹Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
{maggio,leva}@elet.polimi.it

²Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge
{mmaggio,hank}@mit.edu, agarwal@csail.mit.edu

ABSTRACT

The use of control theory is still not common practice in the design and implementation of adaptive computing systems although research into such techniques has been ongoing. The difficulty of developing computing system models suitable for control purposes may be one factor limiting the adoption of control techniques in computing systems.

In this paper we define an equation-based model for the problem of resource allocation that allows synthesis of a regulator to control the phenomenon of interest. In this presentation we limit the scope of this allocator to the distribution of CPUs to running applications in an operating system, and propose a solution based on the Application Heartbeat framework.

Specific attention is paid to the control constraints imposed by the obviously finite number of CPUs. Some solutions are proposed to rescale the number of cores assigned to each application when the demand is not feasible. These solutions are implemented in the Linux operating system by means of useful tools and libraries. Simulations and experimental results support the analysis.

Categories and Subject Descriptors

D.2.10 [Design]: Methodologies; I.2.8 [Problem Solving, Control Methods, and Search]: Control theory

General Terms

Algorithms, Design, Performance

Keywords

Resource allocation, Feedback control, Design approaches

1. INTRODUCTION

In recent years, the computer science community has devoted increased attention to self-adaptive computing systems, in both the software [14] and hardware domains [15]. These systems are capable of adapting their behavior and resources thousands of times a second to automatically decide the best way to accomplish a given goal despite changing environmental conditions and demands. To realize these self-adaptation loops, different decision mechanisms are often considered: heuristic solutions, machine learning and control theory based ones. Control theory has been widely adopted to build adaptation loops [4, 3, 10, 5]; however, full exploitation of the control theoretical potential is yet to be seen and some limitations have been discussed [6]. Moreover, many contributions are limited to the web server domain, which has some peculiar characteristics that general purpose computation often does not expose, see e.g. [16, 12, 1, 11].

Control theory can be used to allocate resources to a single application [13, 9], where the feedback signal is the application heart rate, measured with the Application Heartbeat framework [8]. In this paper we overcome some of the limitation of the mentioned approach. Specifically, we take a single resource, the number of CPUs assigned to each application, and improve the control system to work with multiple applications. Moreover, we introduce an antiwindup mechanism that rescales the number of assigned cores to each of the instrumented applications according to the machine workload and the applications' performance levels and goals. However, none of the proposed and implemented five solutions outperforms the others in every possible condition. Our suggestions and conclusions are that a priority based mechanism should be used when real time guarantees are essential and it is possible not to accept an incoming task or application to maintain the desired performance levels (heart rates) for the other jobs present in the system. In the case of a less strict constraint on performance levels and when one wants to accept all the incoming requests, a centroid based solution may be preferable. Both these solutions are extensively described in Section 2 and examples of their use are proposed in Section 3. Section 4 concludes the paper.

2. CONTROL DESIGN

The objective of the study is to design a *control* loop for the problem of a single resource allocation, in this case, CPU allocation. More precisely, if the computing system has n CPUs our control system should decide how to allocate these computing units to m applications, based on the desires that each application specified. We will start by presenting a control solution for the case of a single application and show this solution at work with both simulation and a real implementation. Subsequently, we extend this control system to manage multiple applications, distinguishing between *instrumented* and *non-instrumented* applications. An instrumented application directly expresses its goals and provides a measure of its current performance level for the *feedback* phase of the control loop. Quite intuitively, we can control only the performance of the applications that provide us measurements of their progresses. We let the other running applications be managed by the operating system without acting to modify their performance levels.

The m considered applications are instrumented via the Application Heartbeat framework [7] to emit a performance level signal, in the form of *heartbeats*. At significant points of the application's execution, it emits a heartbeat to signal progress toward its goal. Moreover, the application can specify a desired performance level in the form of a desired heart rate. Suppose the application is a web server and it emits a heart beat when a request is served, then the desired performance level corresponds to the throughput of the server. If the application is a video encoder it can be instrumented to emit a signal every time a new frame is encoded, therefore the performance level is the encoding rate. All the PARSEC benchmarks [2] were instrumented via the Heartbeat framework, as described in [8] (notice that this paper contains also the Heartbeat API) and we use these applications for the experimental phase and the model synthesis. Notice that if we want application level control, we should use application level sensors, as done in this work with the Heartbeat framework. In the following, we synthesize a deadbeat controller, sketching out the methodology for a single application to make the paper self-contained and then extending it to the multiple applications case. The methodology of the controller realization is extensively described in [13] and is based on a preliminary modeling phase followed by the *regulator* design. In other words, based on a model of the application, we develop a control system constraining the closed-loop transfer function to have a desired shape. The performance of the application at the k -th heartbeat is modeled as

$$r(k) = \frac{s(k)}{w(k)} + \delta r(k) \quad (1)$$

where $r(k)$ is the heart *rate* of the application, $s(k)$ is the relative speedup applied to the application between time $k - 1$ and time k , and $w(k)$ is the *workload* of the application. The term workload means the time between two subsequent heartbeats when the least amount of resources are given to the application, while the term $\delta r(k)$ is a disturbance introduced to account for non nominal behavior. The controller computes the speedup to be applied and then transforms it into a number of cores to be allocated to the application. An antiwindup mechanism is set not to exceed the number of cores of the machine (and obviously one could not assign a

negative number of cores to an application). The requested action is performed through the `taskset` Unix command. More details on the controller synthesis can be found in the cited reference.

2.1 Single application controller evaluation

To validate the controller, first, one should resort to simulation. Simulating the behavior of the system and the controller saves time by detecting errors early in the controller design phase. Obviously, simulation in this case means assuming the model is correct and simulating the behavior of the controller in that case. Simulation is also useful to test the reaction to some inaccuracies in the model. Figure 1 depicts the simulation of a single application that emits 200 heartbeats. In the first 50 beats the application behaves exactly according to the proposed model, the workload provided as an estimation to the control system is correct; notice that in the case of a non correct but still constant workload, the time to convergence increases but the system still reaches the set point. In the portion of simulation between beat 50 and 100 the workload estimation is still correct but we introduce a medium random disturbance to test the disturbance rejection. In the area between beat 100 and beat 150 we remove the previously introduced disturbance and deal with workload variation, supposing the real workload of the application is time varying and has some heavy oscillations. Finally, in the last part of the simulation, both the workload is varying and the disturbance is introduced, although, both these two non nominalities' magnitudes are smaller than previous tested modeling inaccuracies. The results of the simulation tell us that we should not be concerned much about disturbances but workload variations are hard for the control system to manage. As a matter of fact, we augmented the controller with an identification block, its task being the detection of variations. The identification mechanism and the subsequent adaptive controller are not the topic of this work and therefore no more details are given herein. It is, however, worth simulating the system behavior because this gives us the opportunity to understand the controller's weaknesses and to synthesize a remedy.

Simulation is not enough to verify the controller behavior. To show that the gathered simulation data are reproducible and representative of a real environment, Figure 2 reports a test where a real application is instrumented and executed together with the proposed controller. The controlled application is `bodytrack`, from the PARSEC benchmark suite [2]. The picture shows both the window rate (the average of the heart rate over the last 20 beats), used for control purposes, and the instant rate (the instantaneous application behavior). Notice that the control requirement is to obtain a window rate of 3 heartbeats per second and we are executing the controller without any identification mechanism for the workload, setting its nominal workload to 0.5 seconds. These parameters were set to reproduce comparable results with respect to the proposed simulation of Figure 1. In the real environment¹, the execution duration is 105 seconds. The controller assigns the application 5 to 6 cores out of the 8 of the machine, when the system is in the steady state, with an average power consumption of 148.487 Watts

¹All the experiments reported are run on a Dell PowerEdge R410 server with two quad-core Intel Xeon E5530 processors running Linux 2.6.26.

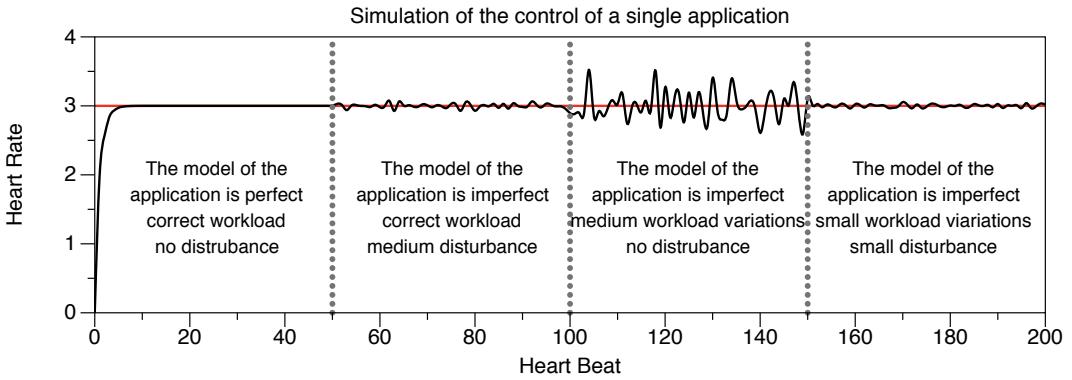


Figure 1: Simulation of a single application control with different model inaccuracies.

(minimum 137.000 Watts, maximum 152.300 Watts, standard deviation 5.103 Watts) and an energy consumption of 15591.100 Joules

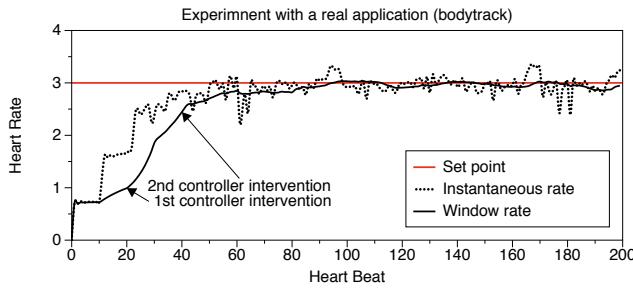


Figure 2: Execution of a controlled single application in a real environment.

When a single instrumented application is running in the system, the problem is therefore solved. In that case, if the workload is not overestimated and if the system is able to meet the target performance, i.e., the application is not requiring more than the real machine capabilities, the measured heart rate reaches the desired one. However, if multiple applications are running in the system, it may happen that the applications will jointly require more CPUs than the number available and the decision on how to assign limited resources to each application will be addressed in the next section.

2.2 Multiple application extension

An intrinsic limitation of the mentioned approach arises when multiple applications have to be managed at the same time, and obviously this is a limitation of every resource allocation mechanism working with finite resources. In this case, the number of cores available in the system is limited to n . With our feedback control strategy the system is completely *decoupled*. In such a decoupled system all the control signals (e.g., the number of cores each of the m applications should have to hit its performance target) are computed independently.

To overcome the mentioned limitation an antiwindup mechanism can be introduced. Its presence means that the control system computes a vector $c(k)$ of m elements, each element $c_i(k)$ representing the amount to be given to the i -th ap-

plication and this vector needs to be rescaled if the sum of its elements exceeds the machine capabilities. To start we can distinguish between antiwindup mechanisms that preserves the decoupling property of the system and solutions that introduce an *a posteriori* coupling in the control calculation. None of these two categories is in principle preferred, notice however that if the antiwindup mechanism is applied *a posteriori*, and needs all the vector data calculations, the system coupling may be unavoidable. To clarify, if the antiwindup is applied just in the cases the sum of the vector elements exceeds n , to compute its result the regulator needs all controllers to act at the same time and thus constrains the system. In the following we will sketch some solutions:

- *Fair distribution*: in this case if we have m applications and n computing units, each of the application has an antiwindup maximum value of n/m number of cores. This distribution is completely fair and its implementation can preserve the decoupling property. However, the solution could lead to a waste of resources, in the sense that if the i -th application needs less than its share, these unused resources might have been assigned to the j -th application possibly allowing it to meet its performance goal.
- *Weighted distribution*: this proposal is similar to the previous one, but in this case the cores are not equally distributed among the applications. A weight vector is introduced to define the importance of the application and to distribute the maximum amount of resources each one could consume. This solution suffers from the same defect as the previous one, but introducing weights allows one application to be favored and may obtain better results for applications with unequal resource needs. Still, the risk with this approach is penalizing a single application while leaving other resources unused.
- *Priority distribution*: in this case, the highest priority process picks the number of cores that it desires, the second highest can use some or all the remaining cores and so forth. If an application has no available resources it is penalized. This mechanism couples the system, all the control actions need to be computed simultaneously and a priority value is needed to distribute the available cores.
- *Core sharing*: with this solution, we choose the number of cores limiting it to the capabilities of the machine. We therefore suppose we can allocate all the

cores of the machine to each independent application, disregarding the presence of other instrumented processes. This means binding each application to some (or all) computing units and relying on the operating system scheduler to schedule the tasks on each of the assigned cores. The main advantage of this solution is that it is easy to implement because it decouples the control actions of independent applications; i.e., the control action for any application can be executed without knowledge of control for other applications in the system. The main drawback is that the behavior is not a priori predictable and one may lose part of the benefits of using a control-theoretical solution.

- *Centroid antiwindup:* suppose you have two applications to be assigned the CPUs; the space of possible solutions is the area of the triangle composed by the three points $(0,0)$, $(n,0)$ and $(0,n)$. It is possible to define a centroid point allowing the weighting of different applications. For example, to give the same importance to the two applications running, the centroid point will be $(0.5n, 0.5n)$. Obviously a more complex weighting system may be introduced according to the relevance of the individual applications. If the computed control point is outside the triangle, the controller selects the point which stands in the line that connects the computed value and the centroid, therefore preserving the relative speedup assigned to the applications. The generalization to multidimensional space, where each dimension is an application to be controlled, is straightforward. The centroid point in that case should be the point in space identified by the vector $(w_1 n, \dots, w_m n)$ where $w_1 \dots w_m$ are the weights of each application and sums to one. This solution intuitively couples the system but it allows preservation of both priority metrics, defined by the weights, and the natural ratio between the computed control signals.

None of these solutions seems to fully solve all the issues introduced by the limited amount of resources, but they represent an interesting *scenario* to test the system behavior and offer some interesting properties to be studied.

3. EXPERIMENTAL RESULTS

In this section we will show results both from the simulation of the proposed allocation policies and from real applications running on a machine. Notice that in the following we will use the term Integral of the Squared Error (ISE) to refer to the quantity

$$ISE_a = \frac{1}{N} \sum_{k=1}^N [\overline{hr}(k) - hr(k)]^2 \quad (2)$$

where a is the considered application, N is the number of data points collected, $\overline{hr}(k)$ is the desired heart rate and $hr(k)$ is the measured heart rate at time k .

3.1 Simulation results

Due to space limitations, we do not show a simulation example where all the mentioned techniques are able to cope with the applications needs. However, it worth noticing that there are cases where the introduction of a coupling mechanism, being it the priority distribution or the centroid an-

tiwindup is useful. Suppose we have a machine with eight cores and four running processes, three of them requiring just one core and the fourth asking for four of the remaining five cores. This situation is feasible, but the fair distribution mechanism would limit the number of cores available for the fourth application to two. Depending on the weight factors, the same situation could or could not be feasible with the weighted distribution described and core sharing may fail in assigning the same core to more than one application and obtaining unpredictable effects due to context switches. The introduction of a coupling mechanism (a priority distribution or the centroid antiwindup) provides a partial solution this problem.

We report data for a single test case where four applications are simulated on a eight-core machine. As for the test case presented in Section 2.1 each application emits 200 heartbeats, with the same disturbances and time variances, i.e., workload variations. The nominal workload of each application is set to 0.5 seconds and the applications' set points are set respectively to 3, 8, 5 and 4 heartbeats per second. Since showing a plot for each antiwindup configuration is space consuming, we report the ISE defined in (2) for each of the applications. There is a randomness for a particular simulation, and we account for this by repeating the simulation [TODO: HOW MANY TIMES?] for each distribution and reporting the average result. In the case of the weighted distribution, the weight vector is set to $[0.3, 0.5, 0.1, 0.1]$, while in the priority case the priority list is set accordingly, privileging the second application over the first, the third and the fourth.

Method	ISE_1	ISE_2	ISE_3	ISE_4	Sum
Fair distribution	0.02	16.05	1.01	0.03	17.11
Weighted distribution	0.02	0.11	11.55	5.75	17.43
Priority distribution	0.02	0.12	0.05	15.16	15.35
Core sharing	0.10	11.97	0.49	0.27	12.83
Centroid antiwindup	0.21	2.28	1.34	0.81	4.64

Table 1: Simulation results: Integral of the Squared Error for a four running applications test case with different antiwindup mechanisms.

Notice that the fair distribution penalizes the second application, which needs more cores to meet its performance target, while letting the others achieve their goals. Given the workload and the simulated machine, in this case, every application is able to get two cores and therefore to emit four heartbeats per second on average. The second and the third application are not able to match their performance levels. The weighted distribution privileges the second application, whose weight is set to be the highest and penalizes the last two applications, whose weight are lower. In the priority distribution case, the only application that suffers from the distribution is the last one, the lowest priority one.

To simulate the core sharing case, we assume that the capability of each core is equally distributed among the applications scheduled for that core, disregarding the context switch time and assuming each application to be treated equally by the operating system scheduler. If all the four applications are running on the first core, the effectiveness of the core is 25% with respect to the nominal one. If just three of them are running on the second core its effectiveness is 33%

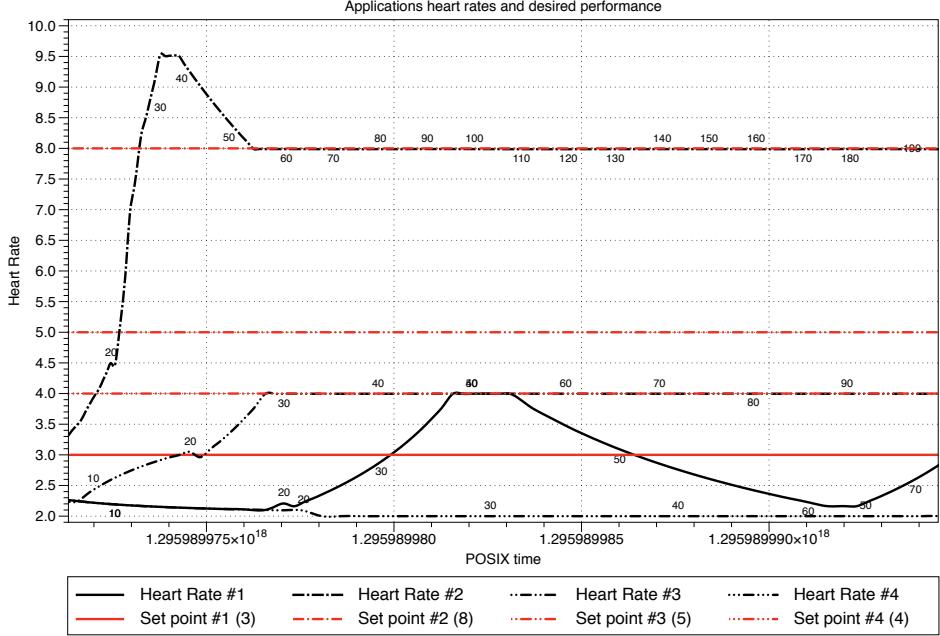


Figure 3: Simultaneous control of the execution of four applications in a real environment with the centroid antiwindup mechanism. The curves are labeled with the correspondent heart beat. The 1st application heart rate oscillates around the desired performance level. The 2nd application meets the level. The 3rd and 4th are a single heartbeat per second below the goal.

of the original one. To keep the control system completely decoupled, it is not made aware of this performance degradation and adjusts the number of cores over time based on the feedback signal.

The *centroid* for the centroid antiwindup case is set according to the weights to $[0.3, 0.5, 0.1, 0.1] \cdot 8$ being therefore the point identified by the coordinates $[2.4, 4.0, 0.8, 0.8]$. The sum of the obtained errors is lower than the one simulated with other solutions, which seems to suggest the centroid antiwindup could be better in exploiting the machine capabilities and balancing the (sometimes competing) concerns of meeting individual application's performance goals while maximizing overall system throughput.

3.2 Real world test case

A real test is here presented, performed on the same eight-core machine as the one proposed in Section 2.1. Four applications, i.e., four instances of the synthetic application described in [13], are launched at a certain time with the same heart rate set points of the simulation and each of these applications emits 200 heartbeats. Notice that this scenario does not exactly replicate the simulation test case, as in a real system application start and stop times have greater variability. Since the set points are different, in fact, just a portion of the execution is simultaneous. This portion of execution time, in which all the four application are active in the system, is shown in Figure 3 when the controller is augmented with the centroid antiwindup mechanism. The implementation of each of the antiwindup mechanisms is still preliminary, therefore we do not code exactly the mechanisms proposed in Section 2.2. The main difference between the simulated techniques and the real ones is that

the control system computes a floating point number of cores (i.e., 2.5), a matter which could be easily solved in the single application case with a time division output mechanism (i.e., assigning 3 cores for half of the control period and 2 for the remaining). In the multiple application case, solving this issue is more complex in that the time division output mechanism has to be applied to the computed vector. In our preliminary implementation the time division output is not applied, a matter deferred to future work, and we just provide the closest integer number of cores with respect to the computed one, which explains the oscillation in the heart rate signal of the first application in figure 3. The absence of the time division output actuation obviously introduced some additional control error and its solution is deferred to future work; however, the test case results still look promising in that the mechanisms seem capable of balancing the needs of the different applications. The real environment is

Method	ISE_1	ISE_2	ISE_3	ISE_4	Sum
Fair distribution	1.58	4.46	1.43	7.97	15.44
Weighted distribution	1.35	3.08	2.20	1.97	8.60
Priority distribution	0.95	0.42	3.00	2.23	6.60
Core sharing	4.69	4.47	14.17	5.11	28.44
Centroid antiwindup	0.47	2.65	1.57	1.24	5.93

Table 2: Experimental results: Integral of the Squared Error for a four running applications test case with different antiwindup mechanisms.

tested and the results are reported in Table 2, which shows the equivalent of the ISE values of Table 1. Some considerations are in order. In this example, the antiwindup mechanism is needed only when all four applications are active in the system. As soon as the first application terminates all the others are able to meet their target performance with-

out rescaling. In that execution portion, the error decreases for each of the remaining applications, although not going to zero. This is due to the lack of the time division output mechanism. Notice, in particular, the numbers for the priority distribution and the centroid antiwindup mechanism. With the priority mechanism the error is distributed among the lower priority applications and the higher priority ones meet their performance levels. On the contrary, with the centroid solution, the error is distributed among the applications more uniformly, penalizing each application according to both its performance level and its needs. This suggests that for real time environments, where one could refuse some tasks to have a feasible workload for the hardware, the priority distribution could be the appropriate solution. In a more general purpose environment, however, the centroid antiwindup mechanism could be preferable to maintain some balance among the requirements of each instrumented application and their performance levels. Combinations of the two approaches could be envisioned to provide a better overall performance metric.

4. CONCLUSIONS

In this paper we used powerful tools from the control domain to design and implement a control system to distribute computing units to running applications in a multi-application multicore environment. In so doing we demonstrated the usefulness of modeling, analysis and simulation to develop a control system, especially in the case the controlled object is a computing system. Since some complications are due to the limited amount of resources, multiple solutions to rescale the number of cores assigned to each application when the demand is not feasible were proposed and implemented in the Linux operating system, relying on the Application Heartbeat framework to measure the performance of each application. Some simulation results as well as some execution traces in a real environment were shown.

5. REFERENCES

- [1] T. Abdelzaher, Y. Lu, R. Zhang, and D. Henriksson. Practical application of control theory to web services. In *Proceedings of the 2004 American Control Conference*, volume 3, pages 1992–1997, July 2004.
- [2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [3] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. Self-managing systems: A control theory foundation. In *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 441–448, Washington, DC, USA, 2005. IEEE Computer Society.
- [4] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
- [5] J. Hellerstein, V. Morrison, and E. Eilebrecht. Applying control theory in the real world: Experience with building a controller for the .net thread pool. *Sigmetrics Performance Evaluation Review*, pages 38–42, 2009.
- [6] J. L. Hellerstein. Why feedback implementations fail: the importance of systematic testing. In *Proceedings of the Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, FeBiD ’10, pages 25–26, New York, NY, USA, 2010. ACM.
- [7] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal. Application heartbeats: A generic interface for expressing performance goals and progress in self-tuning systems. In *Proceedings of SMART10*, 2010. <http://groups.csail.mit.edu/carbon/heartbeats>.
- [8] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC ’10, pages 79–88, New York, NY, USA, 2010. ACM.
- [9] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. SEEC: A Framework for Self-aware Management of Multicore Resources. Technical Report MIT-CSAIL-TR-2011-016, CSAIL, MIT, March 2011.
- [10] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 9–15, Berkeley, CA, USA, 2005. USENIX Association.
- [11] X. Liu, R. Zheng, J. Heo, Q. Wang, and L. Sha. Timing performance control in web server systems utilizing server internal state information. In *Proceedings of the Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services*, pages 75–80, Washington, DC, USA, 2005. IEEE Computer Society.
- [12] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, September 2006.
- [13] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *Proceeding of the 49th international conference on decision and control*, Atlanta, USA, 2010. IEEE Control.
- [14] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009.
- [15] F. Sironi, M. Triverio, H. Hoffmann, M. Maggio, and M. Santambrogio. Self-Aware Adaptation in FPGA-based Systems. In *International Conference on Field Programmable Logic and Applications*, pages 187–192, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
- [16] Q. Sun, G. Dai, and W. Pan. LPV model and its application in web server performance control. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, volume 3, pages 486–489, December 2008.