To appear in *Mathematical and Computer Modelling of Dynamical Systems*
Vol. 00, No. 00, Month 20XX, 1–21

# *A Dynamic Modelling Framework for Control-based Computing System Design*

Alessandro Vittorio Papadopoulos[a] *, Martina Maggio[a], Federico Terraneo[b] and Alberto Leva[b]

[a] *Department of Automatic Control, Lund University, Ole Römers väg 1, SE 223 63 Lund, Sweden*
[b] *Dipartimento di Elettronica, Informazione e Bioingegneria,Politecnico di Milano, Via Ponzio 34/5, 20133 Milano, Italy*

(*2013*)

This manuscript proposes a novel viewpoint on computing systems' modelling. The classical approach is to consider fully functional systems and model them, aiming at closing some external loops to optimise their behaviour. On the contrary, we only model strictly physical phenomena, and realise the rest of the system as a set of controllers. Such an approach permits rigorous assessment of the obtained behaviour in mathematical terms, which is hardly possible with the heuristic design techniques, that were mainly adopted to date. The proposed approach is shown at work with three relevant case studies, so that a significant generality can be inferred from it.

**Keywords:** Computing systems; feedback control; scheduling; memory management; resource allocation.

## 1. Introduction

The complexity of many computing system functionalities is nowadays abruptly increasing [15]. For example, consider the Linux scheduler. In the Kernel version 2.4.37.10 (September 2010) all of its code was contained in a single file of 1397 lines. In version 2.6.39.4 (August 2011) the scheduler code is spread among 13 files for a total of 17598 lines.

Indeed, when such "explosions" are experienced, the overall design approach to the functionalities is to be somehow reconsidered. The presented research proposes to adopt a design approach, entirely based on the systems and control theory. This would allow the reduction of heuristics that are widely present in modern operating (and computing) systems. The main advantage of the abolition of heuristics is that the properties of interest could then be formally assessed [19]. If a control-theoretical design is carried out, the formal tools of stability, observability, reachability and so forth can be brought into play, to state that the system will behave as expected in the presence of unpredictable situations and disturbances. It is worth noticing that to the best of our knowledge this approach has not yet

*Corresponding author. Email: alessandro.papadopoulos@control.lth.se

been attempted, i.e., no computing systems functionality has to date been conceived and developed based on a dynamic model of some physical phenomenon to be controlled[1].

The lack of a system- and control-theoretical attitude in the design of computing system components has quite clear historical reasons, see again [19]. For the purpose of this paper, one fact is most important to notice in this respect. While in any other context controlled objects can be modelled based on physical (first) principles, this is not the case for computing systems, because in such systems the "physics" is created by the designer him/herself. This is well exemplified by a famous quote by Linus Torvalds [34], who wrote

> 'I'm personally convinced that computer science has a lot in common with physics. Both are about how the world works at a rather fundamental level. The difference, of course, is that while in physics you're supposed to figure out how the world is made up, in computer science you create the world. Within the confines of the computer, you're the creator. You get to ultimately control everything that happens. If you're good enough, you can be God. On a small scale.'

In the absence of a modelling framework, however, system design (or according to Linus, creation) is invariantly carried out directly in an algorithmic setting, without any means to formally assess its behaviour. As "more physics" is created, the absence of a rigorous dynamic description may thus, sooner or later, pose intractable problems as for its governance: the scheduler explosion just mentioned is a notable example of this trend.

As a consequence, the non system-theoretical *scenario* sketched out above could to date be tolerated, but it cannot be assured that said tolerability will carry over to the future. Rigorous – and possibly simple – modelling frameworks to ground system design upon are becoming a necessity, since there is much to do in this direction before problems exacerbate [10].

The main message this paper wants to convey, is that if one accepts to re-design part of said system – that has been conceived in an algorithmic way – such a framework can be found by (usefully) limiting the model scope to describe the real physical phenomenon on which the addressed aspects of the system behaviour depend. If this is done, surprisingly simple formalisms can be used—a noticeable example indeed of process/control co-design, and in the authors' opinion, a step forward with respect to previous research.

This paper concentrates on the modelling side of the problem, by showing the ideas above at work, extending [25] with an additional and deeper analysis of the examples treated therein, and proposing a novel framework dealing with memory management. Some words are spent on the consequent advantages in terms of system (and control) design, limiting however the depth to sketching out possible solutions, and referring the interested reader to the convenient literature when this is applicable. On a similar front, a comprehensive presentation of the current state of this research, specialised however to the context of operating systems, can be found in [19].

In this paper, the formalism of discrete-time dynamic systems is exploited. An alternative – and in some cases also coordinated – approach for the control of computing system could be based on supervisory control and discrete event systems [29, 35]. However, in the authors' opinion, the modelling effort carried out in

[1] In the scheduler case, to stick to the example, the phenomenon is how the Central Processing Unit (CPU) is distributed among the running tasks. Such distribution depends on control actions, i.e., on the time slice allotted to each task at each scheduler's intervention, and on exogenous disturbances, such as task blockings, resource contentions, and so on.

this paper would greatly simplify the design of the controllers also if supervisory control was the control paradigm of choice, since it provides with more insights in the problem to be solved and in what influences the dynamics to be controlled.

## 2. The quest for a physics

Before going into details, in this section we spend some words to show that many typical problems in the computing systems domain, can be addressed with a general viewpoint by adopting a dynamic modelling framework. In the next sections, the ideas presented below in general, are specialised and declined in some representative examples.

At the very core of any computing system behaviour there is some strictly physical phenomenon. For example, in the case of an operating system scheduler, that phenomenon has the form

$$\text{accCPU}(k) = \text{accCPU}(k-1) + \text{burst}(k) + \text{disturbance}(k)$$

where $k$ counts discrete time instants, accCPU$(\cdot)$ is the CPU time accumulated by a task, burst$(\cdot)$ is the CPU timeslice allotted to the task, and disturbance$(\cdot)$ accounts for any difference between burst$(\cdot)$ and the actual CPU use by the task.

Similar models can be obtained for many other problems. For example, suppose that an application needs to accomplish its task at a specified rate, like a video encoder that needs to process a desired number of frames per second. Suppose that the application speed depends on some resources, like the number of processing units, and these resources are shared with other applications, so that some arbitration mechanism is required to manage them. In such a case, the present state of the application's progress toward its goal depends on the progress state before the last resource arbitration instant, and on the allotted resources at that instant.

In the most general case, the behaviour of a computing system ultimately depends on extremely fine-grained facts, down to the detailed behaviour of any single assembler instruction and electronics transient. This is one of the main difference between modelling computing systems or purely physical objects. The fine-grained physical level is often the only level that can be rigorously defined in computing system modelling.

On the contrary, in physical systems, there is usually a more abstract modelling level. In thermal systems, for example, one can avoid treating fine-grain phenomena (in that case, molecular motions) since there exist suitable macro-physic entities (e.g., temperature or enthalpy) that allow to write rigorous balances (e.g., of energy) to base dynamic models upon.

In the development of computing systems, in addition, no set of "first principles" has *de facto* ever been sought. Sticking again to the scheduling example, *action* policies are typically defined as "give the CPU to the task with the earliest deadline" by foreseeing their effect in some nominal conditions (for a schedulable task pool, doing so there will be no misses). Alternatively, in the control of the application's progress, the *action* policy can be expressed as "give an additional core if the application is too *slow*, remove a core if the application is too *fast*". Apparently, the algorithmic attitude to the problem hinders the possibility to formally address dynamic properties of the system at hand, as it attempts to find a (control) solution without a modelling phase.

In the addressed domain, in other words, there is classically no distinction among the behaviour of the system in the absence of such actions, the desired behaviour

of the same system, and the way actions are to be determined based on the above. There is no evidence of the fundamental elements of a (control-oriented) modelling process.

Deepening the analysis, one may object that many works deal with computing system control, and do use control-theoretical methodologies, see for example [1, 9, 27, 31] and, particularly, [11]. This is true, but virtually all of them take the computing system *as is* and close loops around it (e.g., aiming at a certain CPU distribution by altering task deadlines). Doing so however requires to model the core phenomenon *plus all the "created physics" around it* (e.g., the existing deadline-based scheduler).

In the authors opinion, the presence of such "unconsciously created" physics is a major reason for the complexity of most computing systems' models, at least as far as the ultimate scope of said models is to design parts of those systems in the form of controllers. To circumvent the problem, one should thus in the first place evidence the core phenomenon, i.e., that part of the system behaviour that really relies on physics and cannot be altered. Most often, modelling that phenomenon is enough to describe the system in a view to suitably control it [21, 25]. In some cases, in addition, the so obtained models will be natively (almost) uncertainty-free, making control design and assessment very straightforward. In other cases, there may be relevant uncertainty, or – in other words – some aspects of the system behaviour will not admit a clear physical interpretation. In such cases, the advice is to figure out some convenient grey box description – as opposite to the black box approaches that the literature dominantly presents [11, 27] – based on qualitative considerations on input-output relationships. As will be shown, this approach generally leads to more complex but still tractable models: control design may be correspondingly harder, but still there will be the possibility of a rigorous assessment.

In the following, some examples are shown of how the proposed approach leads to dynamic models of computing system components that can successfully serve the evidenced needs, while being very simple and thus suitable for powerful and rigorous analysis and control result assessment.

## 3. A unified framework for task scheduling

This section shows how the task scheduling in a preemptive single-processor system can be fully treated having as model class that of discrete-time dynamic systems, in some cases even linear and time-invariant. A few words are also spent on the natural attitude of said modelling formalisms to scale up towards, for example, multicore or multiprocessor contexts, where any other modelling formalism and design approach do experience severe difficulties. The reader interested in more details on the problems encountered by literature approaches can refer, e.g., to [28].

Consider a single-processor multitasking system with a preemptive scheduler, preemptive meaning that the scheduler can interrupt the current task and substitute it with another one. Let $N$ be the number of tasks to schedule. Define the *round* as the time between two subsequent scheduler intervention. Let the column vectors $\tau_p(k) \in \mathrm{R}^N$, $\tau_r(k) \in \mathrm{R}$, $\rho_p(k) \in \mathrm{R}^N$, $b(k) \in \mathrm{R}^{n(k)}$ and $\delta b(k) \in \mathrm{R}^{n(k)}$, $1 \leq n(k) \leq N \, \forall \, k$ represent, respectively,

- the CPU times *actually* allocated to the tasks in the $k$-th round,
- the time duration of the $k$-th round,
- the *times to completion* (i.e., the remaining CPU time needed by the task to end its job) at the beginning of the $k$-th round for the tasks that have

a duration assigned (elements corresponding to tasks without an assigned duration will be $+\infty$, therefore allowing for the presence of both batch and interactive tasks),

- the *bursts*, i.e., the CPU times allotted by the scheduler to the tasks at the beginning of the $k$-th round,
- the disturbances possibly acting on the scheduling action during the $k$-th round (for example because one of the tasks release the CPU before its burst has expired or because of an interrupt management amidst the task operation),

where $n(k)$ is the number of tasks that the scheduler considers at each round. In the traditional scheduling policies $n(k)$ is constant and equal to one—an example of aprioristic constraint that in principle can be relaxed, maybe resulting in better performances. Denote by $t$ the total time actually elapsed from the system initialisation.

A very simple model for the phenomenon of interest is then

$$\begin{cases} \tau_p(k) = S_\sigma b(k-1) + \delta b(k-1) \\ \tau_r(k) = \mathbf{1}_{1\times N}\tau_p(k-1) \\ \rho_p(k) = \max\left(\rho_p(k-1) - S_\sigma b(k-1) - \delta b(k-1),\, 0\right) \\ t(k) = t(k-1) + \tau_r(k) \end{cases} \tag{1}$$

where $\mathbf{1}_{1\times N}$ is a row vector of length $N$ with unit elements, and $S_\sigma \in \Sigma$ a $N \times n(k)$ switching matrix. The elements of $S_\sigma$ are zero or one, and each column contains at most one element equal to one. Matrix $S_\sigma$ determines which tasks are considered in each round, to the advantage of generality (and possibly for multiprocessor extensions). Notice that, since $n(k)$ is bounded, the set $\Sigma$ is finite for any $N$.

Several scheduling policies can be described with the presented formalism, by merely choosing $n(k)$ and/or $S_\sigma(k)$. For example

- $n = 1$ and a $N$-periodic $S_\sigma$ with

$$S_\sigma(k) \neq S_\sigma(k-1), \quad 2 \leq k \leq N \tag{2}$$

  produce all the possible Round Robin (RR) policies having the (scalar) $b(k)$ as the only control input, and obviously the pure round robin if $b(k)$ is kept constant,
- generalisations of the RR policy are obtained if the period of $S_\sigma$ is *greater* than $N$, and (2) is obviously released,
- $n = 1$ and a $S_\sigma$ chosen so as to assign the CPU to the task with the minimum row index and a $\rho_p$ greater than zero produces the First Come First Served (FCFS) policy,
- $n = 1$ and a $S_\sigma$ that switches according to the increasing order of the initial $\rho_p$ vector produces the Shortest Job First (SJF) policy (notice that this is the same as SRTF if no change to the task pool occurs, as can be seen in Figure 1),
- $n = 1$ and a $S_\sigma$ selecting the task with the minimum $\rho_p$ yields the Shortest Remaining Time First (SRTF) policy.

The capability of model (1) to reproduce the mentioned policies is shown in Figure 1, in the case of $n(k) = 1$, $N = 5$, and $S_\sigma(k)$ chosen as described above.

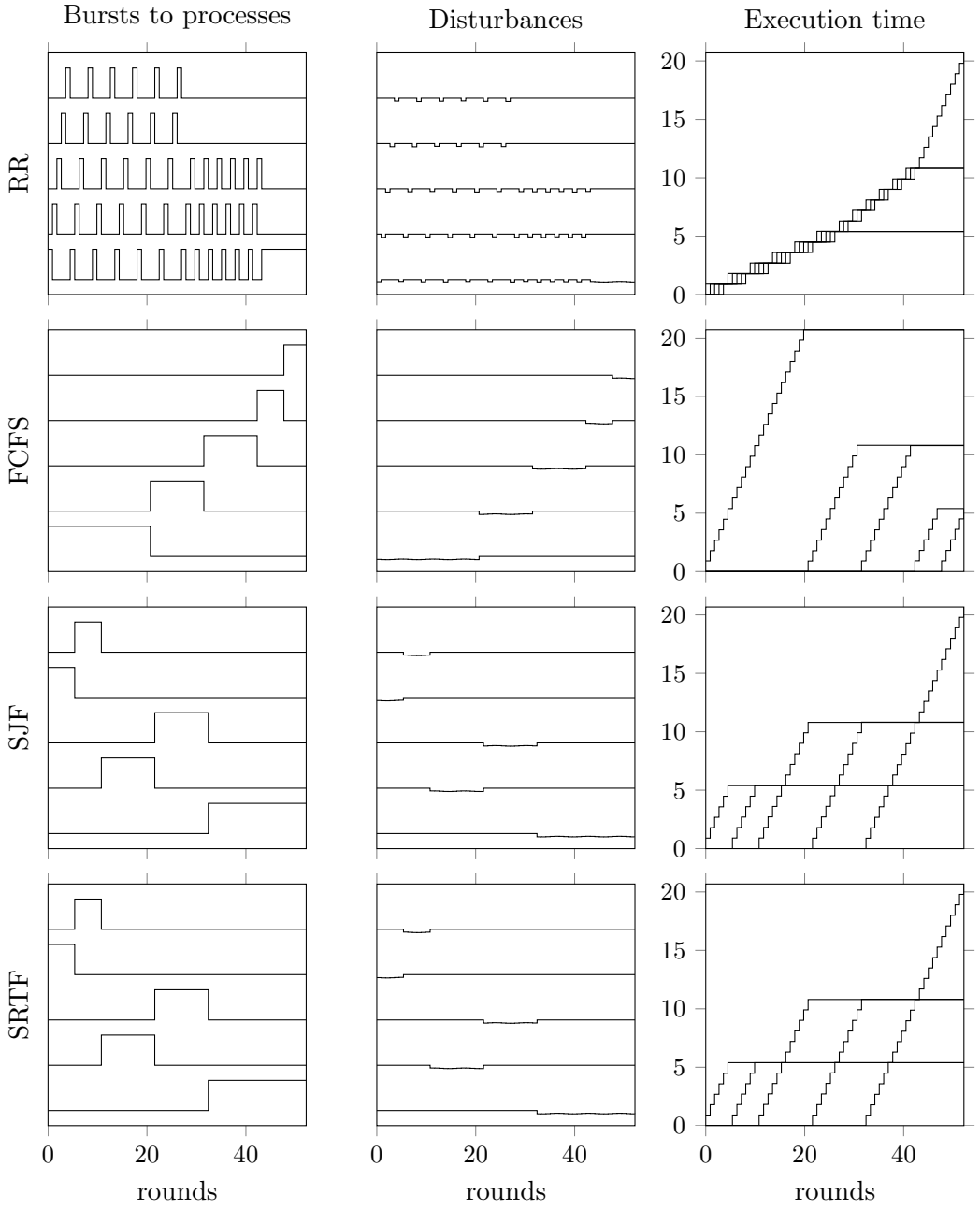


Figure 1. Capability of the presented *single* model of reproducing classical scheduling policies such as RR, FCFS, SJF, and SRTF.

In all these policies, the core phenomenon can be noticed in the form

$$\tau_p(k) = S_\sigma b(k-1) + \delta b(k-1).$$

Also the "added physics" can be noticed, as the algorithm used to select $n(k)$ and/or $S_\sigma(k)$.

If one attempts to model both things together, to close the loop around the existing scheduler, then switching systems must be brought into play.

If, on the contrary, one models the core phenomenon only, and treats all the rest as part of the controller, the single and trivial equation just written is enough. Notice that here in modelling the core phenomenon no uncertainty is present, nor

is there any measurement error, since the only required operation is to read the system time.

Based on model (1) one can thus abandon "non control theoretical" (and often not even closed-loop) choices of $S_\sigma$ as in the examples just sketched, and synthesise schedulers as controllers with very simple blocks, for example of the PI or Model Predictive Control type [18, 23]. The so conceived policies are tendentiously less computational-intensive than those where a control loop is conversely closed around an already functional scheduler, i.e., not around the core phenomenon only.

For example, some work introduce controllers to adjusts the bursts, with the purpose of keeping the entire system utilisation below a specified upper bound [2, 3, 6]. In these works the burst duration is adjusted according to the results of the execution of a controller, built to optimize different cost function. Each of these cost function requires to redesign the control strategy and no control-based selection of the next task is envisioned. On a different page, the authors of [4] re-order the list of tasks to be scheduled with a round robin algorithm in an embedded device, with the aim of reducing cache misses. Control is introduced to meet a system requirement by acting on a parameter of a fully functional scheduler, rather than to simplify the design of the entire scheduler.

The approach proposed here, instead, models the core phenomenon and uses that model to pursue a real control theoretical solution, where properties of the closed-loop system could be formally proved. In this respect, a possibility is to endow (1) with a cascade control structure, aimed at controlling both $\tau_r$, i.e., the round duration, and the distribution of said duration among the active tasks. This can be done with very simple controllers, as shown in [18], and allows to specify the desired behaviour as a certain level of responsiveness, corresponding to the round duration, and fairness, related to the mentioned distribution. The paper just quoted also contains comparisons with some major (non control-centric) scheduling policies, witnessing the advantages of the proposed approach. Note that said approach allows to give a control-theoretical sense to terms like "responsiveness" and "fairness", that are widely used and well understood in the computing systems community.

## 4. A unified framework for memory management

In this section, the proposed approach is applied to another core functionality of operating systems, namely that of memory management.

The operation of a memory manager works can be (roughly) described as follows. The RAM in a computing system is divided in pages of fixed size. Those pages can be allocated to processes running in the operating system, which make memory requests. Of course, the quantity of available memory is limited by the total amount of RAM, and situations in which the processes are requiring more than said upper bound may occur. The memory manager needs to be able to handle such cases, by temporarily saving some pages on disk, i.e., by "swapping" out pages through a specified policy, most frequently the so called Least Recently Used (LRU). Those swapped pages can be requested in a different moment by an application (such an event being termed a "page fault"), and the memory manager is in charge of retrieving the appropriate pages and swap them back in RAM.

The relevance of the problem stems from several reasons. Even if the advances of technology make much more memory available for running applications, these are steadily more demanding in terms of memory pages. Memory is thus still a limited, thus critical, resource. Furthermore, memory is continuously requested and relinquished by processes over time, in an *a priori* hardly predictable manner;

as such, the time scale of memory usage – thus management – is quite fast.

The LRU-based "traditional" attitude toward memory management, sketched out above and dating back to works like [7, 14, 20], has two major issues. One is related to the system-wide nature of the LRU scheme, the other to the purely demand-based (or in other terms, event-triggered) activation of the memory manager. As a result, its behaviour is not optimal in many significant use cases.

A typical example is when a memory-intensive background task is run concurrently with some interactive ones, which can easily happen, for example, when using the same machine for running both heavy batch jobs and a window manager to provide a graphical user interface. When the background task's memory allocations cause the exhaustion of the available RAM, the LRU scheme will swap out pages from arbitrary processes, most probably including the interactive ones, thus causing a significant reduction in their responsiveness. This is caused by the lack of a memory manager that can act on a per-process basis, so as to control *which* are the processes that have exceeded their memory limit, and have to be selected as targets for swap-outs.

The negative impact of swapping out pages onto application responsiveness is a widely known fact, however at present (at least, to the best of the authors' knowledge) no systematic attempts to model the problem have emerged, and only *ad hoc* algorithmic solutions have been introduced.

Another limitation of current memory management systems, as mentioned, is their purely event-triggered nature. A typical example that exacerbates this limitation is when a process transiently allocates a large amount of memory, as it frequently happens for the linking phase at the end of the compilation of large software projects. In this case part of that process will be swapped out, and due to the system-wide LRU scheme, also part of other processes most likely will. When the complex task ends, the memory occupation drops sharply, resulting in a large amount of free RAM. If in this situation the system is left idle, it will not recover responsiveness as fast as it could, due to swap-in being only triggered by application page faults. Therefore, it may happen that memory pages remain swapped out for a long time even if RAM is available. Subsequently, when a process requests those pages, a disk access will be triggered, stalling the process and decreasing its responsiveness. Moreover, the swap-in of those pages may occur when the CPU is highly loaded, while from the swap-out instant till the page faults there may have been plenty of time with a low CPU load.

Summarising, there are two fundamental questions that current memory management schemes fail to address, which are *what* to swap and *when* to do it. The first question addresses per-process memory limits, and could be used to achieve *memory access temporal isolation* [36]. The second question opens the door to transfers between swap and RAM that are time-triggered instead of event-triggered by process page faults. In the opinion of the authors, this is another problem for which hardly any modelling effort has been spent to date, having as result a practically ubiquitous use of pure heuristics, and thus a management (i.e., control) that falls significantly short of perfection. When control-based techniques were applied by closing loops around a memory manager conceived in the traditional way rather than around the core phenomenon alone, the same attitude has often given rise to quite complex solutions, see, e.g., [24].

Analysing the situation with the proposed approach, on the contrary, it is quite straightforward to state memory management as a feedback control problem, by posing for it the following objectives:

(1) use as much RAM as possible without saturating,

(2) give each process a (soft) quota to make it a candidate for swap-outs when this is exceeded (not forbidding anyway its allocation, whence the "soft" adjective above), and
(3) un-swap memory back to RAM when this is possible, based on a time-, not only event-triggered mechanism.

Adopting this viewpoint, here too the core phenomenon physics is quite simple, as the generic ($i$-th) process can be represented by the discrete-time, linear and time invariant model

$$\begin{cases} m_i(k) = m_i(k-1) + a_i(k-1) - dm_i(k-1) + pf_i(k-1) + u_i(k) \\ s_i(k) = s_i(k-1) - ds_i(k-1) - pf_i(k-1) - u_i(k-1) \end{cases} \tag{3}$$

where the state variables $m_i$ and $s_i$ are respectively the quantity of allocated RAM and swap memory. The index $k$ counts the memory-affecting operations, making (3) discrete-time but not sampled-signals. The other quantities are either process-generated requests – i.e., $a_i$, $dm_i$, $pf_i$, $ds_i$, treated here as disturbances – or memory-manager decisions — i.e, $u_i$, that is the input of the model (explained below).

In detail, $a_i$ and $dm_i$ are respectively the allocated and deallocated quantity of memory in RAM, and $ds_i$ is the deallocated memory from swap. The term $pf_i$ represents the page faults that a process can generate (in a highly unpredictable manner, depending on its memory use pattern) and acts symmetrically on RAM and swap.

Note that all the quantities mentioned so far (except for $u_i$) are physically bound to be nonnegative. Also note that all are known by the memory manager, and are therefore measurable without error.

As for $u_i$, this is the only variable on which the memory manager can act, and represents the amount of memory that is moved from RAM to swap or *vice versa*; $u_i$ is thus the only quantity that can take both positive and negative values. The resulting model is composed of two discrete integrators per process, subject to physical constraints, and reads as

$$\begin{cases} m_i(k) \geq 0 \qquad \forall i = 1, \dots, N \\ \displaystyle\sum_{i=1}^{N} m_i(k) \leq \beta \overline{M} \\ s_i(k) \geq 0 \qquad \forall i = 1, \dots, N \\ \displaystyle\sum_{i=1}^{N} s_i(k) \leq \overline{S} \end{cases} \tag{4}$$

where $\overline{M}$ and $\overline{S}$ are the maximum amount of memory and swap in the system, while $\beta$ takes into account that some of the physical memory may be reserved, for example by the operating system itself. The $\beta\overline{M}$ term is here called *global maximum memory occupation*. For completeness, if both memory and swap are exhausted, there is another component of the operating system – named the *out of memory killer* – that terminates processes to free up memory. Such an event is however considered a pathological system condition, indicating a malfunction of some process – that should occur very sporadically – or an erratic swap space

configuration on the part of the system administrator [8]. Both cases are apparently not to be dealt with by the memory manager, thus not addressed herein: for our purposes, in other words, the swap space can be considered infinite.

The physical constraints (4) are partly naturally enforced by the system. For example, if a process has no swap, it cannot generate page faults, and it cannot deallocate memory it does not have (if programming errors causes a program to attempt that, the kernel detects the error and terminates the process before the memory system is set to an inconsistent state).

Memory allocations are conversely unconstrained, hence the system cannot work in the total absence of control, here represented by $u_i(k)$, that is constrained by

$$\begin{cases} u_i(k) \geq -m_i(k) - a_i(k) + dm_i(k) - pf_i(k) \\ u_i(k) \leq s_i(k) - ds_i(k) - pf_i(k) \\ \displaystyle\sum_{i=1}^{N} u_i(k) \leq \beta\overline{M} - \sum_{i=1}^{N} \big(m_i(k) + a_i(k) - dm_i(k) + pf_i(k)\big) \end{cases}$$

Finally, the kernel handles memory in terms of pages, which are a set of contiguous memory locations, with a typical size being 4KB. Therefore, all quantities in the model are expressed in memory pages, and are meaningful only if integers. From such a model, it is quite straightforward to design a feedback controller enforcing the objectives above: details on how to do that can be found in [33].

For the purpose of this work, it is however worth pointing out which are the benefits of the proposed *modelling* approach. First, the problem was naturally split into a *how much* and a *what* part. The proposed control policy decides *how much* to swap in or out and on this basis achieves its goal. Deciding *what* to swap in or out is here irrelevant, and can be devoted to any underlying mechanism without hampering the mentioned achievements.

The same could be stated about resource allocation. Indeed, the authors came to suspect that quite in general, a major reason why simple and powerful control theories seem unnatural to apply to computing systems design, is that problems are formulated in such a way that the *how much* and the *what* part stay intertwined, while when the former is isolated, most often it can be treated with simple formalisms, and normally the results can be realised in a transparent manner with respect to how the latter is addressed. For example, once *how much* to swap out is decided, one can transparently use LRU, possibly on a per-process basis, or any other policy. In one word, there is room for virtually any high-level (i.e., nearer to the software application) policies, pretty much like a well designed layer of peripheral simple controllers eases the setup of more complex, centralised regulations in hierarchical, plant-wide process control.

Finally, notice that here too the model is virtually uncertainty-free, which is quite common a situation in the particular case of computing systems, and definitely worth exploiting with a control-theoretical design approach.

To witness the usefulness of the approach, Figure 2 shows how a model based on (3) can lead to an effective memory management when complemented with the simple control policy described in the following.

- Each process has a memory use limit, the sum of these limits not exceeding the available RAM minus a small amount reserved for the operating system, see (4).
- When the requested RAM exceeds the available one, only processes exceeding

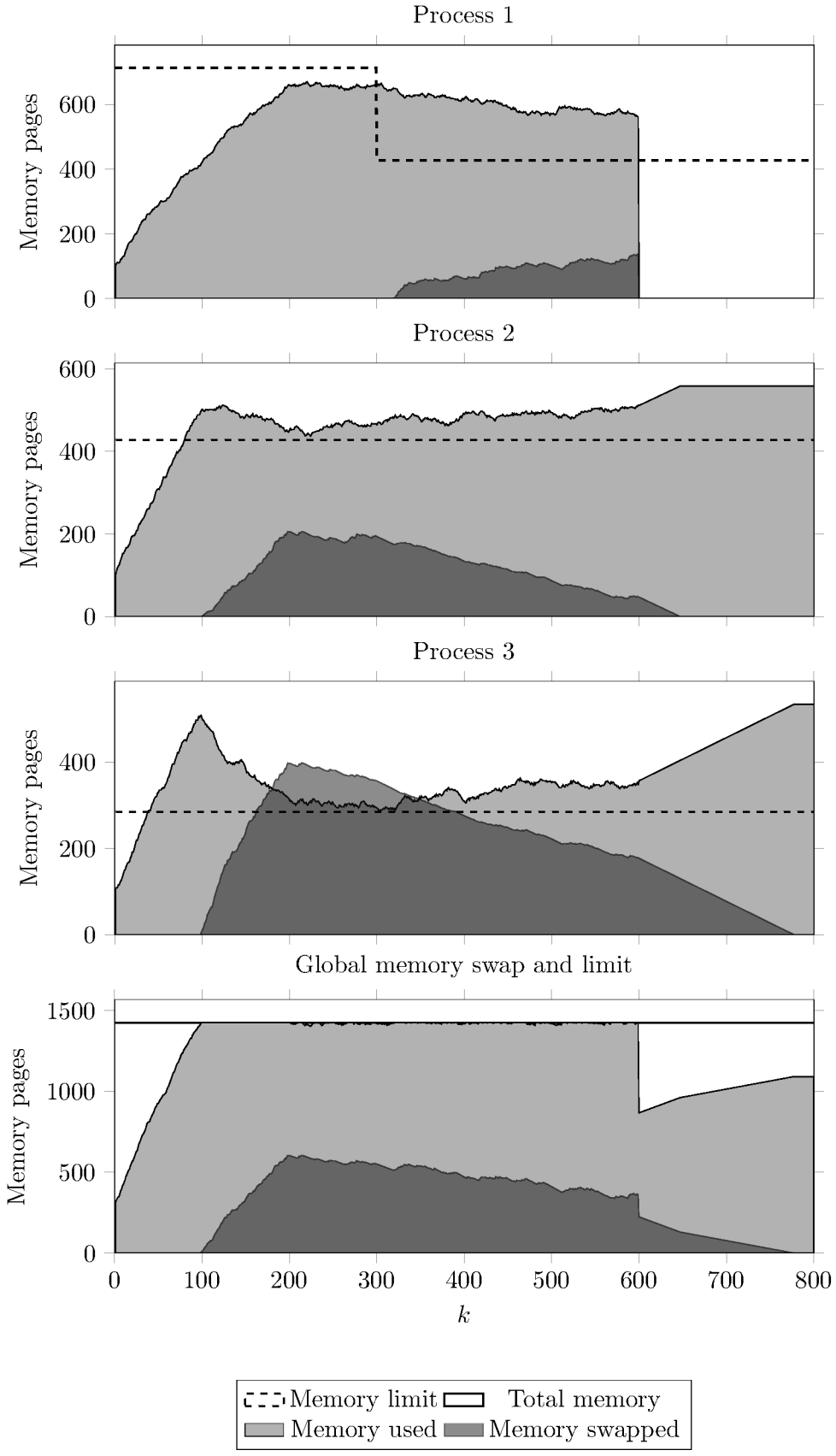


Figure 2. An example of model-based memory management.

their limit can be swapped out.
- When there is free RAM and used swap, a time-triggered mechanism is invoked to swap pages into RAM, based, e.g., on a Least Recently Swapped policy (with obvious meaning).

As can be seen, in Figure 2 there are at the beginning three active processes. All of them allocate memory, until the RAM is exhausted (around $k = 100$). However, since Process 1 is below its memory limit, it is still allowed to allocate RAM, and only the other two are subject to swap-out; only the memory limit change for Process 1 at $k = 300$ makes it too subject to swap-out. Also, when some RAM is available and some swap is used (from $k = 600$), the swap space is emptied by time-triggered swap-in, that co-operate with the (event-based) memory reclaim caused by the termination of Process 1. The interested reader can refer to [33] for further details on the control policy sketched above.

Contrary to the scheduling case, comparing the proposed control strategy to others is not possible here, however. In fact, the present state of the art is practically composed of system-wide policies only. These do not allow any memory usage control on a per-process basis, and therefore one could at most compare aggregate data at the machine level. Independently of "who is the best" in this respect, the reasons why a certain memory management policy could adversely impact the behaviour of a process, reside precisely in the inability of system-wide controls to respond to individual process requirements.

## 5. A unified framework for resource allocation

Up to this point, the focus was set on the management of *specific* resources, namely CPU and memory, for which one could envision *ad hoc* optimisation techniques. In this section, a wider viewpoint is conversely taken, to illustrate how the proposed approach is suited also for the *generic* "resource allocation" problem, that has been gaining a lot of attention in the last years.

In this wider context, the term *resource* may assume different meanings. In a single device, an application may receive computational units or disk space, while in a cloud infrastructure, a resource can be a server devoted to responding to some requests. Each manageable resource is here a *touchpoint* in the sense given to this term in [13]. Some proposals to address the management of a single resource were published in the literature. However, the management of multiple interacting resources is still an open research problem and solutions are more rare [26]. Intuitively, the number of ways the system capabilities can be assigned to different applications grows exponentially with the number of resources under control, and the need for a model is apparent.

In this section we show that, also in the case of resource allocation, a core phenomenon can be identified and modelled. In this case, however, the dynamic relationship between the resource allocated to a system and the performance obtained by the usage of said resources is far from being trivial, and uncertainties are generally present. If one installs additional sensors in the system so as to measure exactly what pertains to the core phenomenon, the resulting models are still much simpler and reliable than those obtained by attempting to describe the system as is.

Generally speaking, the resource allocation problem consists in dynamically modifying the amount of system resources (memory, disk, bandwidth, number of computing units, and so forth) allotted to an application, in such a way the said ap-

plication progresses towards its goal at the desired rate. Recalling the example of Section 2, one may want a video encoder to process exactly 30 frames per second, despite different amount of computational resources needed by the individual frames, and the overall system load. Quite intuitively, the progress rate – that in this work is measured in WorkLoad Units (WLU) per second – is defined on a per application basis (e.g., for a video encoder it could be the completion of one frame).

In most cases, however, a measure of the mentioned progress rate is not available, since usually hardware performance counters are used [17, 32]. The relationship between the progress rate and typically measured quantities is another clear example of added physics – or better, in this case, physics that should not be in the control loop – as the core phenomenon is here "how the progress rate dynamically reacts to resources".

On a time scale suitable for evaluating (and possibly controlling) an application behaviour, the effect of allotting more or less resources to that application, can be viewed as practically instantaneous. However, the efficacy of a given resource on the application progress may vary over time. For example, if an application is presently executing operations that do not require parallelism, the effect of allotting more computational units is modest. Similar considerations hold for memory, disk space, or other resources.

Contrary to the remark above, the time scale of resource-to-performance effects is almost invariantly comparable to that suitable for monitoring and controlling.

Therefore, if one accepts to introduce a progress rate measurement, it turns out that many relevant problems can be treated with discrete time nonlinear dynamic systems of simple structure, obtained with a grey box approach.

For example, when the resources to allot are computational units $c$ and clock frequency $f$ while the application progress rate $pr$ is measured with the Application Heartbeats framework [12], a vast campaign of experiments and data analysis indicated that a model that is simple enough to be used for control but still describes the system in a fairly complete way is

$$pr(k) = p \cdot pr\,(k-1) + (1-p) \cdot (k_c\, c\,(k-1)^{\alpha_c} + o_c)\,(k_f\, f\,(k-1)^{\alpha_f} + o_f) \qquad (5)$$

where parameter $p \in [0, 1)$ is essentially related to the sampling time used for the performance measurements, thus not application specific; the other (time varying) parameters account for resource response of the application. Specifically, $k_c, \alpha_c$ and $o_c$ denote how the application responds to changes in the number of computational units $c$ while $k_f, \alpha_f$ and $o_f$ take care of the responses to clock frequency variations. Note that (5) contains a nonlinear static (multi-)input characteristic cascaded to a linear dynamics, in accordance with the idea that the control time scale is very slow with respect to the actuation one, and complexity resides essentially in the actuators' influence on the process.

Model (5) is apparently of the grey box type, as its structure is envisioned *a priori* based on "physical" considerations, while its parameters come from an identification process.

In fact, in most of the addressed situations [22], parameter $p$ (the discrete-time pole) typically takes low values in the $[0, 1)$ interval, indicating that at the control time scale, the action of actuators is nonlinear but practically instantaneous. Some exceptions may arise for example when some actuating action requires to negotiate resources with the operating system, e.g., posting requests that may be fulfilled at a time scale comparable to that of control, but nonetheless the modelling hypotheses introduced hold reasonably true in all the cases of interest, and in most of them the system to be controlled actually behaves as a nonlinear static one cascaded to

a pure one-step delay.

As a possible objection, application behaviour variabilities can be present and depend on many factors, including for example the processed data, hence the proposed modelling approach may not seem very useful. However, its usefulness can be perceived by observing that, with a sufficiently wide – yet in general affordable – number of profiling tests, one can obtain range and rate bounds for parameter variations. By generating parameter behaviours based on that information, one can then simulate a potentially infinite number of possible application behaviours in much less time than the same number of real runs would require, which is very useful in a view to synthesise and assess controllers. Notice that attempting to do the same thing with classical black box identification applied to linear models – a widely used approach – is for the problem at hand less effective, as such models are structurally inadequate, and any order selection procedure would eventually produce very complex structures.

Needless to say, reverting for a moment to control, the simplicity of (5) – once that model was tested for the capability of actually replicating application runs – suggests correspondingly simple regulators, contrary to what one would conclude based on standard black box models.

Coming to some examples referring to benchmark applications, Figure 3 shows the `bodytrack` and `vips` measured progress rate and the one estimated with the identified simulation model (5) for a particular run, where the parameters' behaviour was obtained by means of an Extended Least Squares procedure. The used applications are `bodytrack` and `vips`, taken from the PARSEC benchmark suite. The *rationale* behind the suite, together with its use, is presented in [5], to which the interested reader is referred for details.

Figure 4 conversely shows the outcome of the classical black box identification process, using the $ARX$ (AutoRegressive with eXogenous input) and the $ARMAX$ (AutoRegressive and Moving Average with eXogenous input) model structures for a run of the `vips` application.

Figure 3 illustrates that (5) is actually capable of replicating the data, by catching main variabilities and trends in a way suitable for control design—its sole purpose here. Figure 4 also suggests that $AR(MA)X$ models are not keen to capture the relevant application behaviour. In fact, if one tries to identify the same data with the Matlab Identification toolbox, performing an order selection for the $ARX(n_a, n_b)$ model, the result is that the identification procedure tries to give to the model as much higher an order as it can, indicating that the *structural* choice is not adequate.

For completeness, the grey box model (5) used in the presented examples, denoting with $\hat{\vartheta}$ the estimated parameter vector, is parametrised for `vips` as

$$\hat{\vartheta}_{\texttt{vips}} = \begin{bmatrix} k_c \\ \alpha_c \\ o_c \end{bmatrix} = \begin{bmatrix} 258.75388 \\ 1.1930687 \\ 681.67218 \end{bmatrix},$$

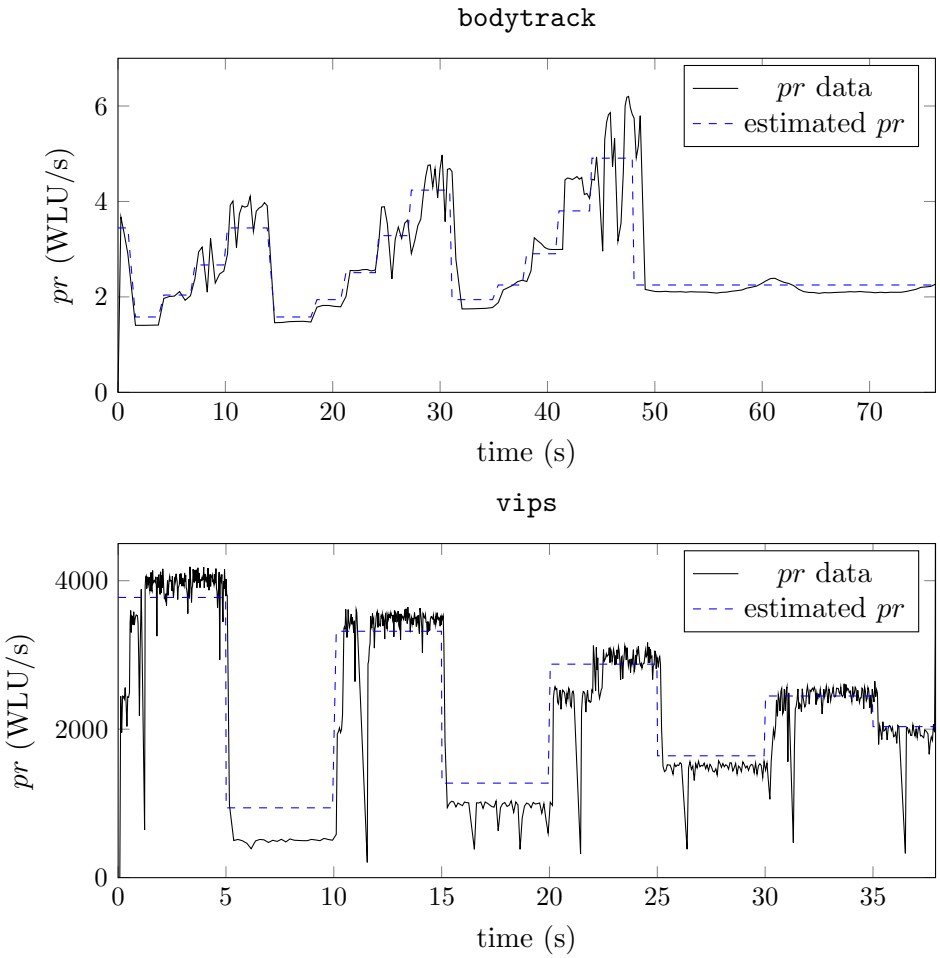


Figure 3. Collected data from the specified software application (black solid line) and simulation with the grey box identified model (blue dashed line).

and for `bodytrack` as

$$\hat{\vartheta}_{\texttt{bodytrack}} = \begin{bmatrix} k_c \\ \alpha_c \\ o_c \\ k_f \\ \alpha_f \\ o_f \end{bmatrix} = \begin{bmatrix} 0.1931659 \\ 1.613834 \\ 3.5964752 \\ 2.3736936 \\ 0.1609101 \\ -1.9965658 \end{bmatrix}.$$

In addition, by introducing a fit measure, the obtained models can be ranked. The fit measure allows to determine how close the output of the estimation is to the real process that it models. Here, the measure is set to

$$\left[1 - \frac{\|Y - \hat{Y}\|_2}{\|Y - \bar{Y}\|_2}\right] \cdot 100$$

where $\hat{Y}$ is the output of the estimators and $Y$ is the measure of the real data. Table 1 shows the obtained results in the `vips` case.

Notice that, starting from the system insight induced by the grey box model, successful adaptive control could be achieved with an $ARX(1,1)$ structure.

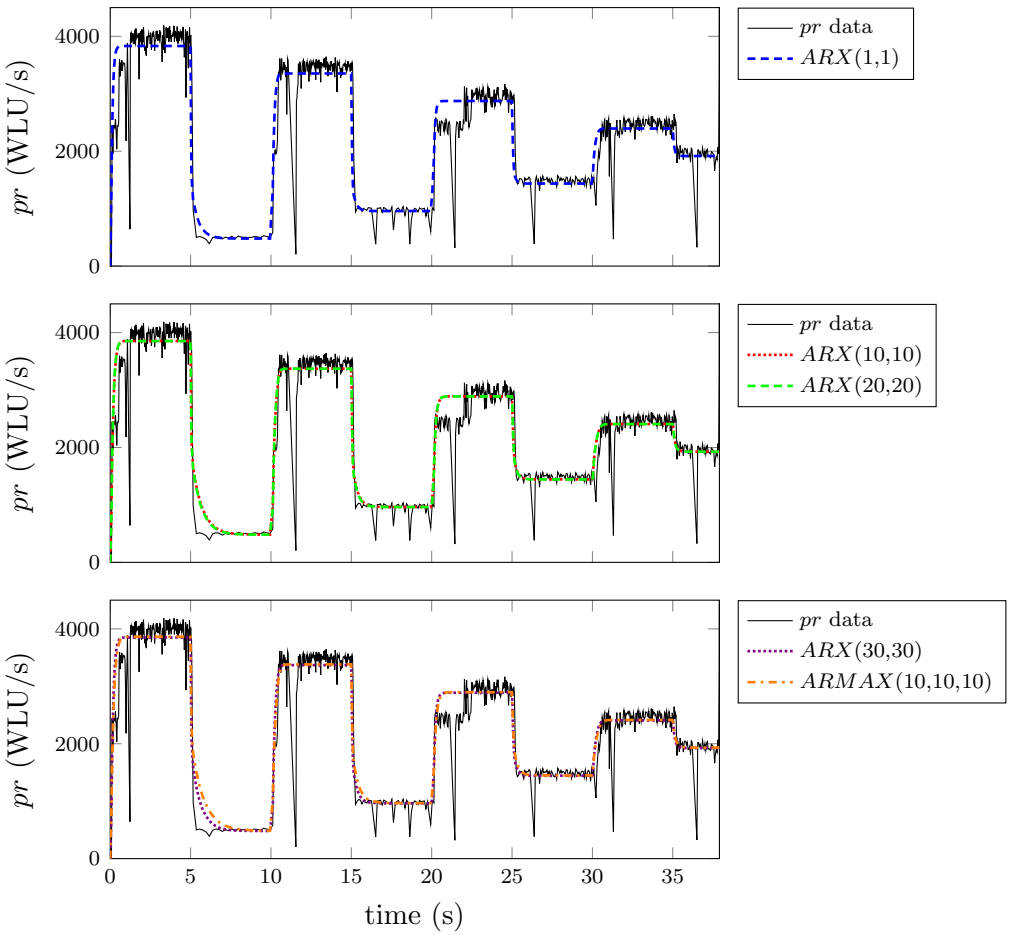


Figure 4. Identification results for the vips software application with different model structures. The data used for the identification are denoted in black with a solid line, the simulation results of the $ARMAX(10, 10, 10)$ in orange with a dashed-dotted line, the $ARX(30, 30)$ in violet with a densely dotted line, the $ARX(20, 20)$ in green with a densely dashed line, the $ARX(10, 10)$ in red with a densely dotted line, and the $ARX(1, 1)$ in blue with a dashed line.

Table 1. Results obtained with the Matlab Identification Toolbox for the `vips` application with various model structures.

| Model | Delay | Best Fits |
|---|---|---|
| $ARMAX(10, 10, 10)$ | 1 | 62.24 |
| $ARX(30, 30)$ | 9 | 61.63 |
| $ARX(20, 20)$ | 9 | 61.53 |
| $ARX(10, 10)$ | 9 | 61.36 |
| $ARX(1, 1)$ | 1 | 58.98 |

To end this section with some control-related material, an example is presented on what can be achieved in that respect. Figure 5 shows experimental results performed on the same real applications, i.e., `bodytrack` and `vips`, when their progress is regulated by an adaptive predictive controller based on model (5).

As can be seen, the required set point is well attained also in the presence of application behaviour's variations, thus proving the effectiveness of the underlying modelling approach.

It is worth mentioning that in the resource allocation literature, using mathematical models for the allocator design is not the typical case, and heuristics or reinforcement learning – model-free – techniques are preferred [22]. This is an

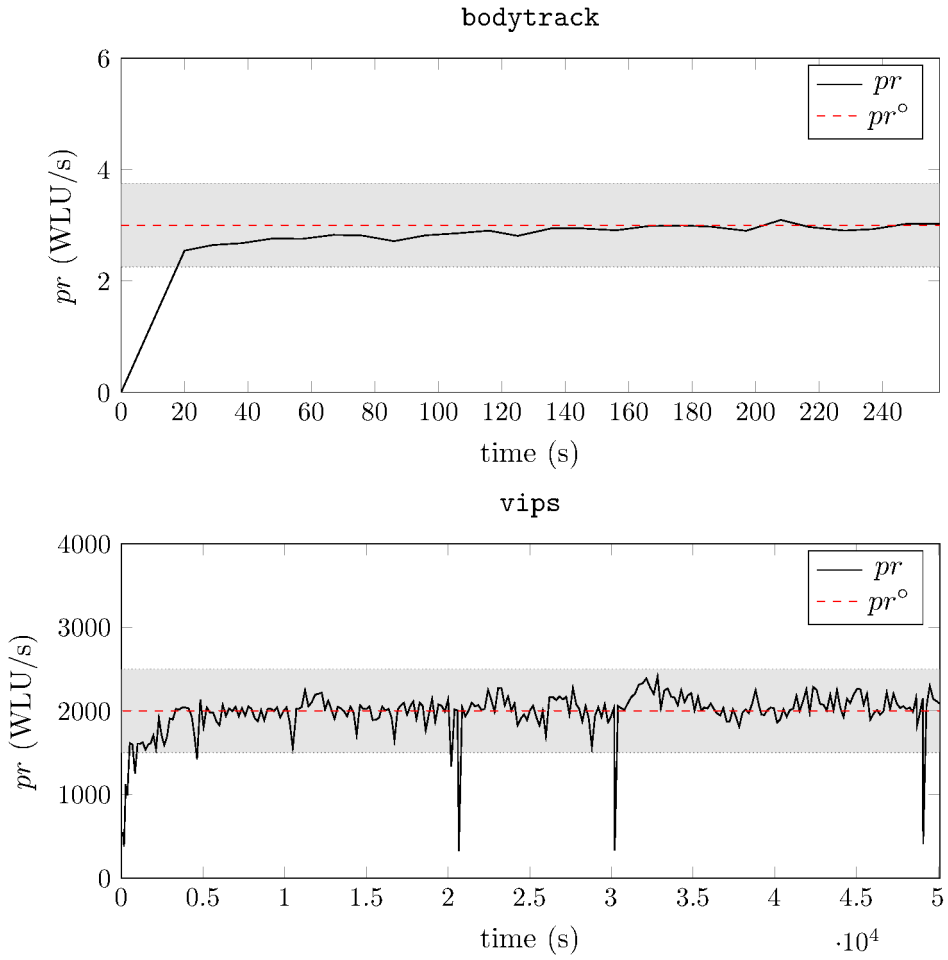


Figure 5. Experimental control results with `bodytrack` and `vips`: the application progress rate is required to attain a specified set point value.

acceptable approach whenever the problem at hand is quite complex or difficult to manage, but if a (relatively small) effort in the modelling phase is spent, many model-based control techniques can be adopted obtaining much better results. Figure 6 shows the results that can be obtained with `vips` controlled with different techniques [22], i.e., a heuristic one (top row), a State-Action-Reward-State-Action (SARSA) algorithm (second row), an adaptive control scheme (third row) and a Model Predictive Control (MPC) technique (bottom row). Those techniques were designed to manage both the Single Resource (SR) case and the Multiple Resource (MR) one.

As can be noticed, in the case of model-free techniques, the achieved performance are worse than in the case of model-based ones, evidencing that spending some effort in the modelling phase can significantly improve the control results — recall that the experiments were conducted on real applications. It is also evident that in the Model Predictive Control example, the controller adapts to exploit the presence of multiple resources, and will need more time to accomplish its task. On the contrary, using a single resource (SR) greatly improve the convergence rate. However, it has to be noted that the multiple resource (SR) controllers tend to settle in states that are more power hungry than its multiple resource counterparts. However, minimising power consumption falls outside the scope of this paper and should be investigated more.

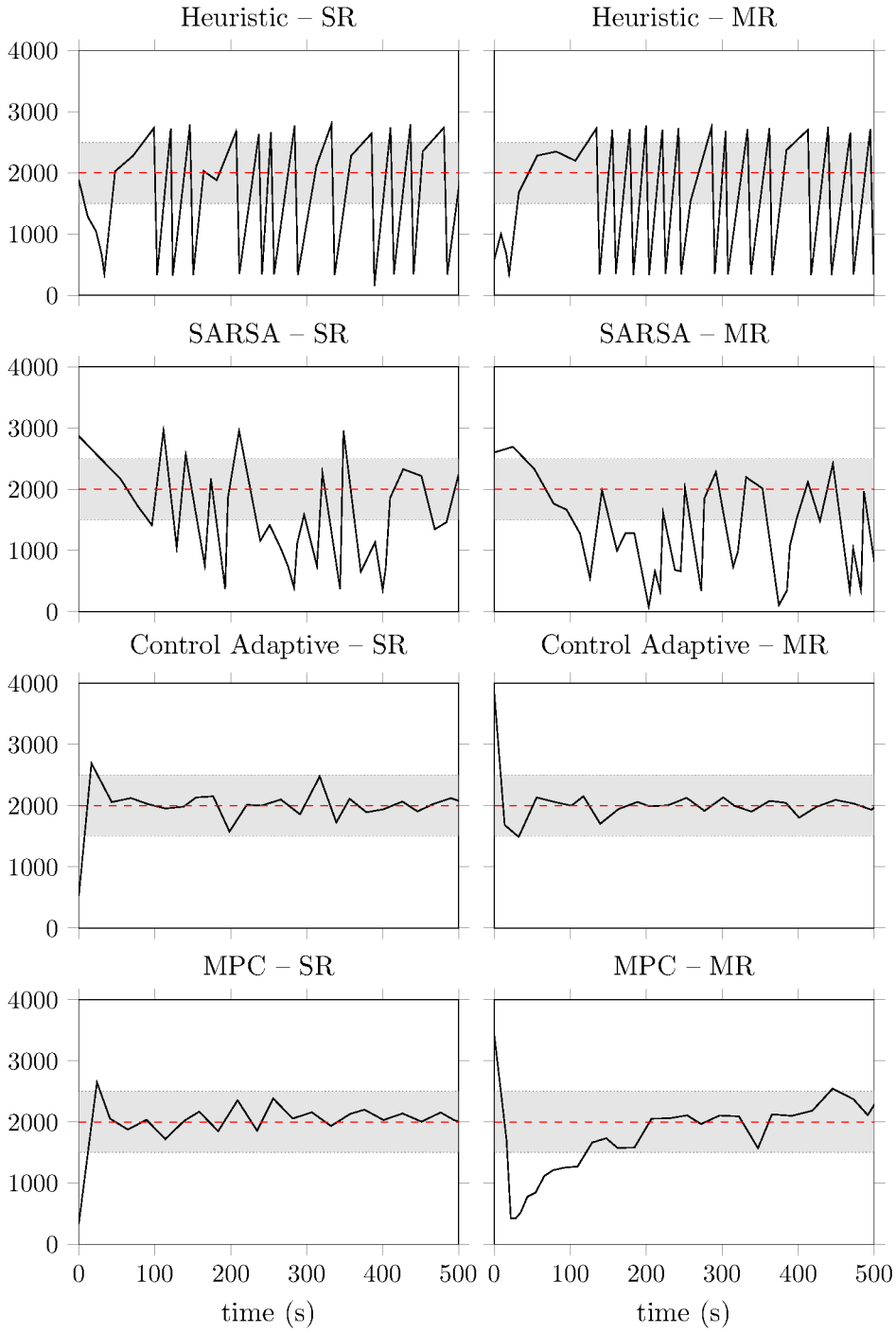


Figure 6. Experimental results with different model-based/non-model-based techniques.

## 6. Retrospect and future directions

We have presented different case studies on the use of discrete-time dynamic models for a control-oriented design of computing system components. It is now the time to collect and organise the so gathered experience, and make some general statements.

If one approaches computing systems for modelling – and possibly control – purposes, the paradigms that naturally appear most keen to be applied are undoubtedly event-based. A vast literature is available on the use of queue networks,

automata, and the like [16, 30, 31]. In our opinion, this is less general than one may think at a first glance. There are cases – probably more than those mentioned herein – where it is more convenient to identify some phenomenon that can be described with discrete-time models, and proceed accordingly as here exemplified. Quite frequently, the involved models are simple, sometimes even linear ant time-invariant. Such simple models (think again to the linear case) allow for a rigorous analysis of stability, reachability, controllability, and similar structural properties.

Furthermore, this simplification is also based on the idea of using discrete-time but not (necessarily) sampled-signals models. One could well say that this is a viable way of dealing with events by using a modelling paradigm with more powerful design and assessment tools.

Finally, since we aim at modelling *phenomena* occurring in computing systems more than *components* of those systems, the approach naturally leads to address both control and design problems—or more precisely, to view the matter as process/control co-design.

The considerations above allow to establish quite deep a relationship between modelling and control for computing systems, and for other application domains like industrial processes. Sticking to this example, it is well known that some phenomena and control objectives are best dealt with with time-driven models, while others call for event-based ones. Correspondingly, a vast literature is available on how to structure a complete (process) control strategy comprising both time- and event-based elements, coordinated in a view to attaining the general objectives for the problem at hand. We suggest that the same approach be applied to computing systems, as attempting to stick to only one of the two mentioned paradigms quite often makes the problem hard to tackle, and often limits the achievable results.

## 7. Conclusions and future work

This work proposed a novel approach to the modelling of computing systems. The main idea behind this approach is to capture the relevant dynamics of computing systems with the simplest possible models, grounded on some "physical" principles. The approach was shown at work with three case studies, and some general ideas were drawn from that experience.

Along this research line, future developments can be foreseen as the application of the presented ideas to other computing system problems, like for example bandwidth allocation. Much further work is required, but an innovative attempt was here made to circumvent one of the main obstacles for co-design success. This attempt is possibly a starting point to rethink from scratch core functionalities of computing systems with a model-based and control-theoretical attitude.

## Acknowledgment

This work was partially supported by the Swedish Research Council (VR) for the projects "Cloud Control" and "Power and temperature control for large-scale computing infrastructures", and through the LCCC Linnaeus and ELLIIT Excellence Centers.

## References


[1] T. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23:74–90, 2003.

[2] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 71–80, 2002.

[3] B. Alam, M. Doja, and K. Biswas. Finding time quantum of round robin cpu scheduling algorithm using fuzzy logic. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 795–798, 2008.

[4] K. W. Batcher and R. A. Walker. Dynamic round-robin task scheduling to reduce cache misses for embedded systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 260–263, New York, NY, USA, 2008. ACM. .

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[6] G. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23:7–24, 2002.

[7] W. Chow and W. Chiu. An analysis of swapping policies in virtual storage systems. *IEEE Transactions on Software Engineering*, 3(2):150–156, 1977.

[8] J. Corbet. 2.6 swapping behavior. `http://lwn.net/Articles/83588/`, May 2004. [Online; accessed 03-June-2014].

[9] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. A control theory foundation for self-managing computing systems. *IEEE journal on selected areas in communications*, 23(12):2213–2223, Dec. 2005.

[10] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey. Fulfilling the vision of autonomic computing. *Computer*, 43(1):35–41, 2010.

[11] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, Sep. 2004.

[12] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. 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*, pages 79–88, New York, NY 10036, USA, 2010. ACM Press.

[13] IBM. An architectural blueprint for autonomic computing. Technical report, Jun. 2005.

[14] R. Jones. Factors affecting the efficiency of a virtual memory. *IEEE Transactions on Computers*, 18(11):1004–1008, 1969.

[15] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36 (1):41–50, 2003.

[16] M. A. Kjær and A. Robertsson. Analysis of buffer delay in web-server control. In *American Control Conference (ACC), 2010*, pages 1047–1052, Jun. 2010.

[17] R. Kufrin. Measuring and improving application performance with PerfSuite. *Linux Journal*, 2005:4–10, Jul. 2005.

[18] A. Leva and M. Maggio. Feedback process scheduling with simple discrete-time control structures. *IET Control Theory & Applications*, 4(11):2331–2342, Nov. 2010. .

[19] A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo. *Control-based Operating System Design*. IET Control Engineering Series. IET, London, UK, Jun. 2013. ISBN 978-1-84919-609-3.

[20] H. Levy and P. Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 18(3):35–41, 1982.

[21] K. Lindqvist and H. Hjalmarsson. Identification for control: adaptive input design using convex optimization. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 5, pages 4326–4331 vol.5, 2001. .

[22] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambro-

gio, A. Agarwal, and A. Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(4):36:1–36:32, Dec. 2012. ISSN 1556-4665. .

[23] M. Maggio, A. V. Papadopoulos, and A. Leva. On the use of feedback control in the design of computing system components. *Asian Journal of Control*, 15(1):31–40, Jan. 2013. ISSN 1934–6093. .

[24] E. Mumolo and G. Bernardis. A novel demand prefetching algorithm based on volterra adaptive prediction for virtual memory management systems. In *Proc. 30th Hawaii International Conference on System Sciences*, volume 5, pages 160–167, 1997.

[25] A. V. Papadopoulos, M. Maggio, and A. Leva. Control and design of computing systems: what to model and how. In *Proceedings of the 7th International Conference of Mathematical Modelling, MATHMOD'12*, volume 7, pages 102–107. IFAC, Feb. 2012. .

[26] A. V. Papadopoulos, M. Maggio, S. Negro, and A. Leva. General control-theoretical framework for online resource allocation in computing systems. *IET Control Theory & Applications*, 6(11):1594–1602, Apr. 2012. ISSN 1751-8644. .

[27] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 33–42. IEEE, 2012.

[28] M. Pinedo. *Scheduling Theory, Algorithms, and Systems*. Springer, third edition, July 2008.

[29] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[30] A. Robertsson, B. Wittenmark, M. Kihl, and M. Andersson. Design and evaluation of load control in web server systems. In *American Control Conference, 2004. Proceedings of the 2004*, volume 3, pages 1980–1985. IEEE, 2004.

[31] M. Shor, K. Li, J. Walpole, D. Steere, and C. Pu. Application of control theory to modeling and analysis of computer systems. In *Proceedings of Japan-USA-Vietnam Workshop on Research and Education Systems*, 2000.

[32] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, Jul. 2002.

[33] F. Terraneo and A. Leva. Feedback-based memory management with active swap-in. In *Control Conference (ECC), 2013 European*, pages 620–625. IEEE, Jul. 2013.

[34] L. Torvalds and D. Diamond. *Just for fun: The story of an accidental revolutionary*. HarperBusiness, 2002.

[35] W. Wonham and P. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988.

[36] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.