How control-friendly is a computing system? And how control-friendly could it be?

Alberto Leva, Silvano Seva, Federico Terraneo [*]
Alessandro Vittorio Papadopoulos [**] **Martina Maggio** [***]

[*] *DEIB, Politecnico di Milano, Italy*
(e-mail: {alberto.leva,silvano.seva, federico.terraneo}@polimi.it)
[**] *Mälardalen University, Västerås, Sweden*
(e-mail: alessandro.papadopoulos@mdh.se)
[***] *Automatic Control Department, Lund University, Sweden*
(e-mail: martina.maggio@control.lth.se)

Abstract To date, not that much. Improvements are possible, but some system-theoretically grounded re-design is necessary. We discuss the matter based on our experience, and as a consequence, we come to distilling some design clues and research directions.

Keywords: Control in computers; Control-based computing system design.

1. INTRODUCTION

The idea of applying control to computing systems dates back more or less to when networks were about to spread—the TCP protocol was specified in Cerf et al. (1974), where incidentally the term "Internet" was coined. As an example of that pioneering research, Agnew (1976) employed differential equations to model congestion and to manage it through optimal control.

The boost of networks then led to more and more complex services, and besides to manage traffic among servers and clients (Gafni and Bertsekas, 1984; Altman et al., 1999; Jagannathan and Talluri, 2002), the need for control emerged in the servers themselves (Abdelzaher et al., 2003). Also, computational power started migrating toward the edge of networks (Murphy and Mapp, 1996; Raz and Shavitt, 2001), bringing about a more distributed need for resource allocation and control (Li and Nahrstedt, 1999) and in fact paving the way toward the present boost of "as a service" frameworks, cloud computing (Barry, 2003), and the corresponding control needs.

The white paper by IBM (2003) introduced "autonomic computing" to collectively denote the subject, and a huge research *corpus* started that we cannot review herein: the interested reader may begin his/her navigation e.g. from the books by Hellerstein et al. (2004); Janert (2013); Leva et al. (2013) and their bibliographies.

With extremely sporadic exceptions, both the said research *corpus* and the applications show a crisp separation between the design of the computing system and of its control. The former is created according to the principles of hardware/software engineering, essentially without control in mind. Then, if deemed necessary, control moves in.

Since over a decade, our group attempts to complement and sometimes overcome this attitude. Instead of just closing loops around a computing system *as is*, often requiring identification to unravel *a posteriori* the dynamics of a non control-aware design, we concentrate on how to possibly re-design parts of the computing system together with their controls, or in some cases even just *as* controls.

As such, we have been asking ourselves the question in the title many times. In this paper we summarise our findings, and propose some ideas that we deem beneficial.

2. THE MAIN RESEARCH QUESTION

The Cyber-Physical (CP) paradigm applies to any system where a C part (in our context, software running on some computer) interacts with a P part, no matter its nature. In the particular case where the P part is a plant and the C one its control, we talk about "computers for control".

We restrict the focus mainly to this case; in fact the P part could even be for example the visitor of a web site, and the following analysis would still apply. However, having controls as the physical counterpart for general ideas is easier for us, and helps streamlining the treatise.

To realise controls, one has to take the C part of a CP *model*, which is by definition an abstract description of the involved controllers, and turn it into software. Doing so means taking asynchronous objects evolving in parallel – e.g., block diagrams for modulating controls and automata or Petri nets for logic ones – and casting them into some imperative language suitable for a synchronous device with limited parallelism like a set of processors. There are established methodologies to do that, see e.g. Borges et al. (2010). However these methodologies are based on execution models (think of the operating "massive copy" cycle of PLCs) that share a fundamental assumption: *the device running the C part is significantly faster than the dynamics of the P part.*

In addition to governing physics outside the computer, no matter the nature of that physics and the actions to carry out on it, the software has to manage things *inside* the computer. This means that the C part above in fact

relies on another C part – think now for simplicity of the operating system – that is in charge of providing the former with the resources needed to carry out its duty according to the requirements. This is "control for computers"—or, as we are going to show, also and sometimes better "control *in* computers".

Apparently, managing the inside physics is necessary no matter what the software has to do with respect to the outside world. Control in computers is far more widespread than computers for control. But since both are ultimately about using software to govern some physical object, it seems natural that the same theories and methods apply.

In fact, the research we present emerges from observing that the last statement is not completely true. This is due to historical reasons, see e.g. Chapter 1 in Leva et al. (2013), but besides those, primarily to the fact that the second C part we just spotted has its natural P counterpart in the hardware. As a consequence, in this region of the overall system, the above fundamental assumption for realising a C part in software "the control engineer's way" very often does not apply. On the contrary, *the P part is much faster than the C one.*

There are corner cases and exceptions to the *scenario* just sketched, no taxonomy can be perfect. However we believe that these do not diminish the practical generality of the analysis and the design approach we propose, thus we defer their treatise to future more extensive works. Summarising, therefore,

- software sits amid two physical worlds, one outside and one inside the computing system;
- taking software as the C part of the overall system, we have to split this C part into an "outbound" and an "inbound" one;
- the outbound C and the outside P form a CP system, subject to specifications such as response times, Service Level Agreements (SLAs) like responding to the user within a given time span, and so forth;
- the outbound C in general operates much faster than the outside P evolves;
- in the particular case of control, we have well established theories and methods to manage the CP system above, and this is "computers for control";

however, in addition,

- the inbound C has to govern an inside P to provide adequate resources to the outbound C, thus being potentially critical to its correct and efficient operation;
- the inbound C and the inside P form another system, that we call "Physical-Cyber" (PC) and not CP
- to evidence that the inbound C in general operates slower than the inside P functions;
- addressing this particular kind of systems is "control in computers".

The PC nature of the inside system is a fundamental reason – others are discussed below – why computing systems look unfamiliar, and quite often not so friendly, to control specialists.

Our main research question, that we articulate and discuss in the following sections, is therefore to what extent and how we can extend CP-centred control design and implementation methods to the PC case – ultimately targeting the compound PCCP system we just defined – and what to do if this is not possible, or not convenient, or any combination thereof.

3. CONTROL IN COMPUTERS: GOOD & BAD NEWS

We start our analysis by pointing out and briefly commenting some facts about control in computers, ending with the major curses to be faced. We focus on the modulating side of the matter, but should we analyse the logic side as well, we would come to substantially analogous conclusions—if not (possibly) for just a remark given later on in the first part of Section 4. Needless to say, no exhaustiveness is claimed.

3.1 The good news

No measurement errors. Quite often, controlled variables just coincide with their measurements. There is no error in reading the length of a queue, the requests processed in the last second, and so forth; these are just numbers in memory. There is no error either in acquiring quantities like the CPU (Central Processing Unit) time consumed by a task, because the same entity (in the example, the system timer) both prescribes the quantity (e.g., by firing preemption interrupts) and provides its reading *after* the prescribing action is exerted. The same is apparently not true when dealing with variables that require a real transducer, like the CPU temperature or supply voltage.

Actuation immediacy. Once a command reaches an actuator, it is applied and exerts its action immediately, as there is nothing material to move. Once again this may not totally apply to problems like frequency and voltage control, but these are in fact more naturally viewed as electronics, and the involved dynamics (think of a PLL lock) are really almost negligible. In general there can be delays for the command to reach the actuator, however.

Many measurable disturbances. Often, a disturbance to a control loop comes from inside the system (e.g., a controller to prescribe the frame rate of a video application can suffer from a frequency reduction caused by the one that governs the CPU temperature). As such, information about that disturbance is readily available, and often – see above – without measurement error either. There are however exceptions, as shown later on.

3.2 The bad news

Ambiguously defined or complex quality indicators. Besides the good news above about measurements, we must notice that some quantities that one may want to control, though easy to define, are *in nature* difficult to measure unambiguously. For example, the load of a CPU has no instantaneous value; it has to be defined over an interval – typically as the fraction of that interval when the CPU was not idle – and the length of that reference interval has relevant consequences. The same applies e.g. to the throughput of a server, and even more to high-level QoS (Quality of Service) indicators. Also, such indicators are frequently much more complicated than just a quadratic

tracking cost: brutalising for brevity, should one want to achieve QoS directly via Model Predictive Control (MPC), he/she would have to very often incur the difficulties of *economic* MPC (Rawlings et al., 2012).

Layering-induced variable delays. Getting some measurements, even if inherently exact in the sense above, may require invoking hardware/software layers that were not designed to provide any certainty about the time they will take to respond. This is true e.g. for many operating system modules that were originally conceived for monitoring purposes, not to operate in a control loop.

Actuator quantisation. Several actuators are heavily quantised. In some cases this is inherent to their structure – for example, a GPU (Graphic Processing Unit) is not preemptable – and in practice cannot be relaxed. In some other cases however this stems from design choices dictated e.g. by architectural simplification, in fact relaxable, but extremely detrimental to control. For example, some PLLs limit the possible frequencies of a CPU to only three or four values while there is no conceptual reason to do so.

Over-actuation with heterogeneous dynamics and interaction. In general, a controlled variable is influenced by several actuators. For example, one can increase the throughput of a server by augmenting its CPU *quota* in the active VMs (Virtual Machines), or adding new VMs, or both. However the first action can take milliseconds, the second even a minute. In this case there is also a time asymmetry, because from the control viewpoint a VM is removed right from the moment when the purpose-specific processes aboard it are stopped, which is a fast operation. Such asymmetries are not so common, but the interaction and superposition of actuators with very different natures and time scales, is encountered quite frequently.

Time-varying interaction. Constraints may cause control loops to sometimes couple with one another and sometimes not. A trivial example is resource allocation: as long as the sum of all the requests does not exceed the total availability, any loop allotting resource for a given goal does not perceive the presence of the others, while this ceases to hold when the total hits the maximum. Incidentally, the continuous variability of the playing actors' set – cardinality included, think e.g. of the task pool for a scheduler – hampers the use of otherwise quite natural solutions (if not for possible memory footprint issues) like explicit MPC.

Data-originated perturbations. The request of resources (in the broadest sense of the term) heavily depends not only on the running applications, on which for example a scheduler with admission policy has some authority, but also on the data that they are processing (e,g., encoding the video of a conference or a football match are very different burdens). As such, there are non measurable and practically unpredictable perturbations, that can possibly be bounded in amplitude by profiling the addressed system, but not in frequency because forthcoming data, let alone a quantification of their computational weight, are ultimately unknown.

Practical upredictability. An application may transition abruptly and unexpectedly from a certain resource utilisation pattern to another (e.g., from CPU-bound to communication-bound). Of course a well designed predictor (for example, of power consumption) will eventually recover such an event, but the point is that "eventually" may be too far in the future for an effective control. Sticking to the example, it may be feasible to predict CPU power consumption well on say a one second horizon, but it is practically impossible to do the same *reliably* on a millisecond-scale one, which is what one needs for thermal management.

3.3 The three major curses

The variable physics curse. Very often, the sensing and actuation points made available by present computing systems do not touch the real physics directly. One can change priorities and time *quanta* for a scheduler but not force a context switch to some chosen task exactly when desired, one can enqueue a network transmission but not take over the transceiver and send immediately, and many other examples could be given. A strong motivation for this, simplifying for brevity, is found in the concept of *driver* to encapsulate the details e.g. of a peripheral toward upper software layers. But no doubt, cascaded to the *real* physics of the peripheral, a controller sees the "virtual" one of drivers. This virtual physics can be inefficient for control, as already suggested when mentioning layering-induced delays, but there is more. If it is not designed *and maintained* in coordination with control, it can change – for example owing to some new hardware functionality that software people decide to exploit – in such a way to make a previously well performing control unusable.

The hidden inefficiency curse. In a paper titled "The Linux scheduler: a decade of wasted cores" that we warmly suggest to the reader, Lozi et al. (2016) evidenced that the CFS (Completely Fair Scheduler) was unable to enforce the apparently simple invariant "no core without tasks to run while at least another core has more than one in its runqueue" and showed that curing this can sometimes lead to over 100× (sic) speedups (*ibidem*, Table 3). We do not discuss their solution here, but rather strongly point out that in a definitely fundamental component of the operating system as the scheduler, so simple an invariant was violated, hence so far was the system from its achievable performance, *and for years nobody ever noticed.* In other control domains we have methods to quantify how far a system is from its theoretical optimum. In computers this is in general not the case, and inefficiencies can stay unnoticed unless their symptoms become evident: an inefficient scheduler does not hang the machine, who knows how fast that machine could optimally run its applications? As the authors of the quoted paper point out, who keeps profiling tools active all the time, and also on the operating system? More in general, care has to be taken – and it may be difficult – to not fight the wrong enemy, and also to not complicate the system for an ultimately inadequate payback.

The stolen time curse. Last but not least at all, in the end control in computers is there to make a system more efficient at running its applications, but the time to compute the control signals is *stolen* from that available to those applications. For example, when requests are processed differently in a web server to fulfill

latency requirements (Klein et al., 2014) the time spent to determine how a request is handled is subtracted from the computation of the response. The achieved improvement must be worth the stolen time, and unless controllers are *really* efficient and lightweight, this can be far from trivial to achieve—for example, but not only, when the remarks made above about MPC apply.

3.4 Wrap-up

When applying control in computers as these are presently designed and maintained, one has frequently to deal with over-actuated systems with time-varying and hardly predictable interactions, subject to abrupt and practically unpredictable perturbations, containing variable measurement delays and crudely quantised actuation, with the goal of enforcing quality constraints often not easy to translate into set point tracking and/or disturbance rejection problems, and with very little time available to possibly apply some optimisation-based approach, or anything else requiring a non negligible computational effort.

Summing up, despite the good news above, the systems to address contain *by design* a collection of the least desirable characteristics a control engineer may happen to come across. And to top, from time to time a new generation of hardware, operating systems and so forth, may come along and revolutionise the *scenario* very rapidly.

Definitely, not so friendly a setting. However through years of experience at least we now have a characterisation of the issues to face, and most important, the evidence that many of them really came into the *arena* only *by design*. Hence we believe that computing systems could be made much more control-friendly than they are to date, by adopting a PCCP viewpoint and focusing on a control-grounded design of the inbound C part.

4. AN ALTERNATIVE VIEWPOINT

The key to the perspective shift we propose, is to carry out any design having in mind what is physically inevitable and what is not, as in the computing domain thisis in general less obvious than in others. In a nutshell, our experience in this respect led us to conclude that

(1) the PCCP abstraction fits the "control in computers" context;
(2) the outer CP part can do control in the most classical sense, but also carry out some "computer only" task like computing a batch job within a deadline—conceptually there is no difference;
(3) no matter the specific purpose of the computing system, "governing it to do its (CP) job properly" ultimately means controlling the inner P part with a well designed inbound C;
(4) the inner PC part originates most of the difficulties in applying the control design techniques we are familiar with, and that we here denoted as "CP-centred";
(5) this is because in the PC part *as presently designed and maintained*
    (a) P is often faster than C, as already said,
    (b) contrary to practically any other control context, *the physical/cyber and the process/control partitions normally do not coincide* because of the evidenced "virtual physics",
    (c) and the bad news and curses we pointed out are there;
(6) a control-grounded (re-)design of the inbound C is therefore necessary, and the considerations made so far prove this necessity, because in the absence of such a design
    (a) the system as seen by the outbound C tends to elude any first-principle modelling, if not to appear "not governed by any laws of nature" (Årzén et al., 2006, p. 11) [1] ,
    (b) crudely, one ends up tackling – e.g., by identification – modelling errors and uncertainties that are in fact but the result of somebody else's previous design.

In addition, as we shall see in this section,

(1) many inbound C design problems lend themselves to being designed as controls, typically as *cascade* controls, admitting however to *replace* (not control) part of the virtual physics;
(2) however this does not solve the problem completely when the "P faster than C" characteristic plays a relevant role, and in that case
    (a) either the affected control functions can be moved to hardware, circumventing the issue and allowing one to continue using CP-centred control design in the sense of Section 2,
    (b) or some new control design paradigm needs devising.

We would like to stress once again the importance of *redesign*. There are many works, also on CP systems control, that evidence the need for some resource management coordinated with the control task, see e.g. Lindberg and Årzén (2010). However they generally tend to preserve the virtual physics in the non control-grounded inner C (e.g., the paper just quoted considers acting on scheduling parameters instead of replacing the scheduler) and do not appear to aim at a systematic, problem-agnostic approach to the said re-design.

We finally conjecture that the "faster P" issue mainly involves logic controls, for example when the need is to govern a peripheral that is itself designed (and often also specified in the documentation) as a state machine. Should our conjecture be verified, the need for new paradigms would substantially – or at least, to a large extent – amount to devising controller execution models not based on the PLC massive copy cycle approach. The matter is at present under investigation.

5. APPLICABILITY AND GENERALITY

A very natural question, at this point, is how general the proposed design viewpoint is. In this respect to date we have no exhaustive answer, but we are definitely confident.

[1] For the sake of completeness, the quoted sentence continues with "at least not on the macroscopic level". We consider this to support our approach based on a control-grounded inbound C, viewed in this case as a means to conveniently act at a somehow "micro" scale with respect to the outer CP system.

First, we learnt that in some cases setting up a well designed inbound C is simply a control problem in nature: one has just to isolate and model the controlled phenomenon in a way that is convenient for feedback control design. In these cases there is no need to re-formulate anything, and the only reason why the present inbound C layer is not a controller is that the people who designed it do not know control, or did not apply it properly.

This first category of cases clearly comprehends "physical" controls like the CPU temperature, where the design deficiency just noticed is evident: compare for example the simple solution by Leva et al. (2017) with the non control-based alternatives in Kong et al. (2012), or with not so simple proposals like Wang et al. (2009), or focusing on applications, with the Linux Thermal Daemon (Intel Corporation, 2014). But quite interestingly, very similar considerations apply also to not so "physical" controls. A notable such example is time synchronisation: Terraneo et al. (2014) showed that with a proper modelling, the problem is nothing but a disturbance rejection one for a discrete-time LTI plant.

Other cases – the majority, we have to admit – are not so straightforward to address as controls. For these, a certainly indirect but reasonable way to answer the question of this section, is to see how abstracted a problem can be made, while preserving contact with some concrete instance, for the viewpoint and its consequences on design to still apply. We now attempt such an exercise.

A vast category of control problems in computers either ultimately reduce to controlling some processing rate, or in fact sit atop controlling such a rate. The former case occurs when the main requirement is meeting stipulated deadlines, transferring data at a required speed, keeping the pace of some input/output device or application (Hoffmann et al., 2010), processing samples timely, and so forth—all variations over one theme, see for example Klein et al. (2014). The latter case occurs when the requirement is more articulated, for example abiding by a deadline that needs splitting into intermediate ones for parts of the overall task (Baresi et al., 2016), or guaranteeing maximum average response times on some horizons, both depending on the contract with the client who originates the request requests, and so forth; in such situations there will most likely be some high-level intelligence to manage the problem, but this will need to rely on something to be as fast as expected, hence will more or less explicitly compute speed set points. We do not treat this entity and its possible natures herein, but once it has accomplished its task, the obtained subproblems fall in the first category.

Furthermore, it is obvious that if we had an infinitely fast computing system with infinite resources, we would not experience any of the problems mentioned so far, and as a consequence, that for each of those problems we could in principle figure out a progress rate (hence implicitly a resource management to permit it) above which the problem disappears. Therefore, if we could control progress rates reliably, the task of the intelligence just mentioned would be greatly simplified no matter its nature.

This said, for space reasons, we illustrate our viewpoint with reference to an abstract case which we generically name just "progress" control.

The speed of progress dynamically depends on the applied "thrust", possibly on its present value, and on disturbances *not related to "resources"*, like e.g. the variable computational weight of data. In turn, thrust depends on some "command", but also on the available resources, and in practice this relationship is almost invariantly instantaneous, hence algebraic. Finally, the availability of resources varies – in general dynamically – according possibly to their present state, to *all* the commands that turn into their utilisation, and to other exogenous disturbances such as the temporary unavailability or the reduced power of some computing unit. Summarising the above *in abstracto*, we can represent the controlled "plant" as

$$\begin{aligned} \frac{d\ progress}{dt} &= f_P\big(progress, thrust, disturbances\big) \\ thrust &= f_T\big(command, resources\big) \\ \frac{d\ resources}{dt} &= f_R\big(resources, commands, disturbances\big) \end{aligned} \tag{1}$$

and depict its role in the overall control system, according to present design practice, as in Figure 1.

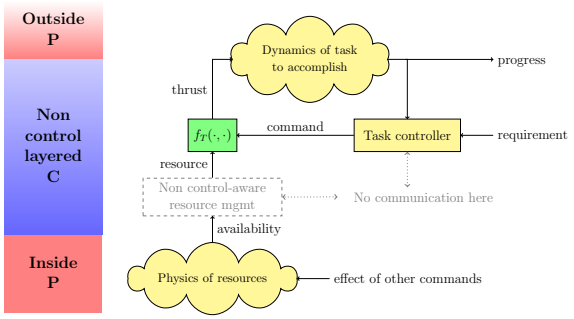

Figure 1. PCCP system without control-grounded layering of the C part.

Observing the so obtained control system evidences what we consider – at least as long as applications based on processing rate control are addressed – a crucial problem: *either there is nothing to smooth out the effect of competing resource requests so as to ensure that the commanded thrust is actually exerted, or if such an entity exist, it is designed in such a way that obtaining a dynamic model for it is extremely cumbersome.*

As a consequence, both the task of deciding how much thrust to command, and that of ensuring that the issued command really results in that thrust, are left with the same entity, labelled "task controller". Said otherwise, we have some virtual physics (the dashed grey box) aboard what (the C layer) we would like to be just control—that is, the design of the C layer starts out with constraints that one would probably not introduce if allowed to apply control to just the real physics (that of the resources). We can therefore confirm that the problem is not distinguishing the inbound and the outbound C parts, i.e., not enforcing a *control-grounded layering* of the system.

The proposed viewpoint ultimately amounts to enforcing such a control-based C layering by viewing the inbound C as the inner loop of a cascade structure, as suggested in Figure 2. The idea is in fact very intuitive. There is a goal (that of the applications) and a controller (the task

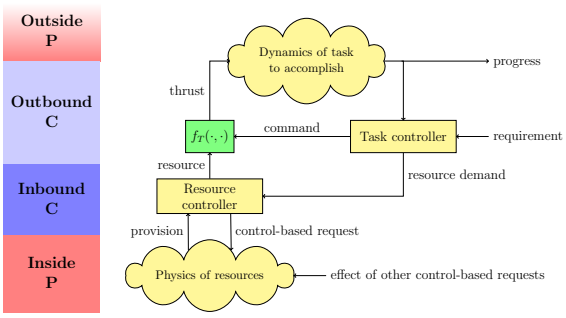

Figure 2. PCCP system with control-grounded layering of the C part.

one) targeted to achieving that goal; this controller emits commands whose effect depends on resources; these are in fact requested and not necessarily allocated as desired, but this is measurable, hence there is some variable responding to the controller command in advance with respect to the ultimate goal, and subject to disturbances. All the ingredients of cascade control are there, and extending the system to reflect the scheme of Figure 2 is natural.

There is not the space here for discussing any example in enough detail, however not even mentioning the possible applications of the proposed idea would be an unacceptable incompleteness. As such, in Table 1 we list just a few declinations of our general proposal to particular examples, broadly ordered by abstraction level. Despite applying the approach to the particular components mentioned could require technologically different operations, the structure of the underlying *control* problems is essentially the same.

To end this brief dicsussion, it is important to notice that the design approach of Figure 1, which we suggest to abandon given its proven shortcomings – in fact corresponds to the mainstream way of introducing *software organisation* layers, where the allocation of resources is managed by some operating system modules with the task of coordinating requests from the applications (most frequently to enforce some "fairness" no matter how defined). However in general these modules "manage the machine" with no awareness of the *needs* of the applications, as drawn in Figure 1, and when some way is provided to the applications for talking to resource managers, this is not done the way one properly designs a cascade structure. Even the simplest consequence of this – i.e., not taking care of making resource management (inner loops) faster enough than the dynamics of requests (outer loops) – often suffices to generate undesired behaviours.

As such, strange as it may seem, given the present design practices for computing system components, the introduction of control-based C layering would be a true revolution—whence the questions that we are going to address in the following.

6. FEASIBILITY, COST, IMPACT

We start with feasibility and cost. According to experience, turning our ideas into software (and sometimes hardware) design is normally quite straightforward. The main problems come from plugging the result into the existing system, and can be broadly divided in two main groups—in the end two further "bad news" in the sense of Section 3.2, but better evidenced at this point.

Poor component isolation. We found no better way to explain this than reporting two sentences by Lozi et al. (2016), who in the "Lessons learned" section wrote

> *The bugs we described resulted from developers wanting to put more and more optimisations into the scheduler, whose purpose was mostly to cater to complexity of modern hardware. As a result, the scheduler, that once used to be a simple isolated part of the kernel, grew into a complex monster whose tentacles reached into many other parts of the system, such as power and memory management.*

Apologising for the crude metaphor, the typical result of such situations – most frequently encountered in low-level components – is that sometimes the surgery turns out to be more invasive than expected. We had such an experience with a control-based scheduler (Leva and Maggio, 2010), that works very well in a POSIX-compliant embedded kernel for microcontrollers (Terraneo, 2008) but we did not yet port into Linux.

An important topic on this front is whether or not "extending the tentacles" was necessary. Here we have to notice that computer people tend to disregard the fact that controllers communicate with one another not only by exchanging signals, but also – and often primarily – through the physics of the controlled process. As such, if phenomenon A influences phenomenon B, this does not necessarily imply that the controllers for A and B must communicate with each other. We do not claim generality, but to date this was by far the most frequent reason for the "unnecessary tentacles" we found.

Unduly stateful layers. When control- and software-grounded layering conflict, co-existence is in principle possible, but can be complicated if the pre-existing layers keep memory of their state and possibly also of the state of the neighbouring ones, or said otherwise, if the set of layers assume that any action on the system occurs only through them. This case is often found at high levels in software hierarchies, but sometimes also involves low ones—we had an experience with synchronisation, see again Terraneo et al. (2014). Here too, avoiding details for brevity, the result is substantially that one has to extend the intervention further than strictly necessary for the originally intended purpose.

Summing up, in general we do not foresee blocking feasibility problems, but sometimes the effort can be unexpectedly significant, and some preliminary cost/benefit analysis should be carried out. Of course things can be totally different when hardware is involved, but to date we only have experience on custom embedded systems where there is enough freedom to make the situation not so different from the software-only case, thus honestly we cannot draw general conclusions; this matter will be further investigated in the future.

Coming to impact – meaning basically on the applications – we have to point out just a single fact. According again to experience, the one (but sometimes not small) difficulty is to take specifications in computer engineering

High-level task to accomplish	Resource to manage	Component(s) requiring a control-based inbound C
Big data applications, web service composition	Containers, virtual machines	Container manager, hypervisor
Website response time control	CPU, memory, disk	Web server, database server
Multimedia streaming	CPU, network bandwidth	Resource manager, network stack
Quality of Service enforcement	Network bandwidth	Network stack
Load balancing	Cross-CPU quotas	Scheduler, resource manager
Real-time control	Single CPU quotas	Scheduler
Virtual memory management	Memory, disk	MMU, Virtual memory and swap
Peripheral bus (e.g.,USB) management	Bus, pipelines	Bus transaction scheduler, drivers

Table 1. A few specialisations of the proposed general design viewpoint.

terms, and translate them into control-compatible ones. One possibility is to preserve the application interface entirely, by adding a translation layer if this is small and agile enough. For example, the scheduler in Leva and Maggio (2010) does not work with priorities, nice numbers and the like, but rather with CPU quotas and round duration set points; it is however possible to create a minimal layer to accept e.g. priorities as inputs. Another possibility is to create a new interface, and managing the co-existence if deemed convenient. Here too, apparently, some preliminary analysis is often advisable.

7. A FEW DESIGN GUIDELINES

Still focusing on modulating control, we can distil some clues for addressing the proposed control-layered design.

(1) Identify the border between inbound and outbound C as precisely as possible.
(2) Relate the result to the existing software layering if there is one, and design interfaces accordingly.
(3) When addressing the inbound C, *do not put in the model of the controlled system anything that is not true physics*: anything but that model must be control, and (re-)designed as control.
(4) When, prior to re-design, some component is connected to some other that from a control viewpoint does not seem to be part of the problem, check if the designers of either or both just disregarded through-process controller intercommunication in the sense above; with a physics-conscious design, the problem very often disappears.
(5) Never mix specifications for the two C layers. The outbound relies on the inbound, but each specification must pertain to one of the two: mixing things up easily leads to poor design.
(6) In the design phase, separate core control functions from sensing and actuation ones; not doing this is an open door to "catering to the complexity of hardware" with the consequences discussed above.
(7) Always assess the designed core control functions in simulation (a matter that we cannot address herein); this helps understanding whether a problem is conceptual – hence appears also with idealised sensing and actuation – or technological. Doing so is another protection against catering to the hardware.
(8) Pay attention to making control-related specifications, parameters and I/O signals clear for the applications atop the designed controls where applicable.
(9) When the stolen time curse is relevant, consider applying event-based control.
(10) Consider hardware/software partitioning of the found solution when, notwithstanding the above, the "faster P" problem persists: it may not be possible at the moment, but it may be possible to emulate it somehow, generally reducing expectations, so as to convince of the usefulness of making it realisable. On this front it is worth noticing that the subject of moving logic controllers written in IEC languages to hardware is receiving interest by research in embedded systems: a recent example concerning very fast and low-power realisations of SFC controllers with FPGAs is Milik and Hrynkiewicz (2019).

These clues do not cure all problems, of course, but we can state that they are not so widely applied, and based on years of experience, that they are surely beneficial.

8. CONCLUSIONS

The growth of control for/in computers is certainly good news. However we bear to gently advise the control community to not consider this subject only as addressing a new application domain, "fascinating because unfamiliar".

Experience convinced us that approaching the matter *only* that way is not likely to unleash all the potential of the control theory, and which is worse, it could deepen the trench between the computer and the control communities. Brutalising a bit, the result can be that computer people carry on designing their systems their usual way – which incidentally seems to entail huge complexity increases in the future – and then control people come in and approach the system with non domain-specific techniques – as specific ones are to date in fact largely to be developed – and more or less as a black box. Doing so, no matter what is obtained, neither of the two involved professionals/scientists learns about the other's culture.

This is not to diminish approaches alternative to ours, of course, but rather to stress that ours should receive attention as well, also and particularly to provide the foundations for tackling more articulated problems. Said

simply and restricting the focus to one example – optimisation – dynamic programming and the like can be fine for the outbound C, but their task is greatly simplified, and their efficacy enhanced, by a control-grounded inbound C.

In addition, we suggest to more in general view control in computers as a process/control co-design and cultural cross-fertilisation *arena*. The former aspect is just obvious. The second would lead to a very effective convergence: computer people learn some control, control people learn about the operation of computers, and all learn together how to design computing systems with control in mind – together with how to create computer-specific control techniques – when this is convenient.

REFERENCES

Abdelzaher, T., Stankovic, J., Lu, C., Zhang, R., and Lu, Y. (2003). Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3), 74–90.

Agnew, C. (1976). Dynamic modeling and control of congestion-prone systems. *Operations research*, 24(3), 400–419.

Altman, E., Başar, T., and Srikant, R. (1999). Congestion control as a stochastic control problem with action delays. *Automatica*, 35(12), 1937–1950.

Årzén, K., Robertsson, A., Henriksson, D., Johansson, M., Hjalmarsson, H., and Johansson, K. (2006). Conclusions of the ARTIST2 roadmap on control of computing systems. *ACM SIGBED Review*, 3(2), 11–20.

Baresi, L., Guinea, S., Leva, A., and Quattrocchi, G. (2016). A discrete-time feedback controller for containerized cloud applications. In *Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 217–228. Seattle, WA, USA.

Barry, D. (2003). *Web services, service-oriented architectures, and cloud computing*. Elsevier, Amsterdam, Netherlands.

Borges, P., Machado, J., Villani, E., and Campos, J. (2010). From SFC specification to C programming language on the context of aerospace systems control. *IFAC Proceedings Volumes*, 43(22), 46–51.

Cerf, V., Dalal, Y., and C. Sunshine, C. (1974). Specification of internet transmission control program. Technical report, RFC 675, December.

Gafni, E. and Bertsekas, D. (1984). Dynamic control of session input rates in communication networks. *IEEE Transactions on Automatic Control*, 29(11), 1009–1016.

Hellerstein, J., Diao, Y., Parekh, S., and Tilbury, D. (2004). *Feedback control of computing systems*. John Wiley & Sons, New York, NY, USA.

Hoffmann, H., Eastep, J., Santambrogio, M., Miller, J., and Agarwal, A. (2010). Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proc. 7th International Conference on Autonomic Computing*, ICAC '10, 79–88. Washington, DC, USA.

IBM (2003). An architectural blueprint for autonomic computing. *IBM White paper*.

Intel Corporation (2014). Linux thermal daemon. [Online] `https://01.org/linux-thermal-daemon`.

Jagannathan, S. and Talluri, J. (2002). Predictive congestion control of ATM networks: multiple sources/single buffer scenario. *Automatica*, 38(5), 815–820.

Janert, P. (2013). *Feedback control for computer systems*. O'Reilly Media, Sebastopol, CA, USA.

Klein, C., Maggio, M., Arzen, K.E., and Hernandez-Rodriguez, F. (2014). Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, 700–711. Hyderabad, India.

Kong, J., Chung, S., and Skadron, K. (2012). Recent thermal management techniques for microprocessors. *ACM Computing Surveys (CSUR)*, 44(3), 13–58.

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

Leva, A., Maggio, M., Papadopoulos, A., and Terraneo, F. (2013). *Control-based operating system design*. IET, London, UK.

Leva, A., Terraneo, F., Giacomello, I., and Fornaciari, W. (2017). Event-based power/performance-aware thermal management for high-density microprocessors. *IEEE Transactions on Control Systems Technology*, 26(2), 535–550.

Li, B. and Nahrstedt, K. (1999). A control-based middleware framework for quality-of-service adaptations. *IEEE journal on selected areas in communications*, 17(9), 1632–1650.

Lindberg, M. and Årzén, K. (2010). Feedback control of cyber-physical systems with multi resource dependencies and model uncertainties. In *Proc. 2010 31st IEEE Real-Time Systems Symposium*, 85–94. San Diego, CA, USA.

Lozi, J., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. (2016). The Linux scheduler: a decade of wasted cores. In *Proc. 11th European Conference on Computer Systems*, 1–16. London, UK.

Milik, A. and Hrynkiewicz, E. (2019). On SFC low power hardware implementation in FPGAs. In *Proc. 16th IFAC Conferemce on Progarmming Devices and Embedded Systems*, 62–67. High Tatras, Slovakia.

Murphy, B. and Mapp, G. (1996). Integrating multimedia streams into a distributed computing system. In *Proc. Multimedia Computing and Networking 1996*, volume 2667, 312–319. San José, CA, USA.

Rawlings, J., Angeli, D., and Bates, C. (2012). Fundamentals of economic model predictive control. In *Proc. 51st IEEE Conference on Decision and Control*, 3851–3861. Maui, HI, USA.

Raz, D. and Shavitt, Y. (2001). Toward efficient distributed network management. *Journal of Network and Systems Management*, 9(3), 347–361.

Terraneo, F. (2008). Miosix embedded OS. [Online] `https://miosix.org/`.

Terraneo, F., Rinaldi, L., Maggio, M., Papadopoulos, A., and Leva, A. (2014). FLOPSYNC-2: efficient monotonic clock synchronisation. In *Proc. 35th IEEE Real-Time Systems Symposium*, 11–20. Rome, Italy.

Wang, Y., Ma, K., and Wang, X. (2009). Temperature-constrained power control for chip multiprocessors with online model estimation. *ACM SIGARCH computer architecture news*, 37(3), 314–324.