A control-theoretical approach to thread scheduling for multicore processors

Alessandro Vittorio Papadopoulos, Roberto Carone, Martina Maggio and Alberto Leva

***Abstract*— Feedback control has been applied to computing systems, usually taking a designed system and closing a loop to adjust some of its parameters. However, the design of computing systems components as controllers have shown advantages with respect to state-of-the-art techniques, especially in the scheduling domain, where uniprocessor schedulers have been designed as discrete-time control structures. However, the most recent computing devices (from smartphone to personal computers) have more than one core and the devised techniques cannot be applied to this context directly. This paper provides the necessary foundation to address the multicore scheduling problem as a control problem, as an extension of the uniprocessor case. We qualify the quantities to be measured and used as feedback signals for tackling the extension. We also present some control solutions and compare them using a simulator, publicly available to foster the research on the topic. The comparison shows that the devised policies have low computational complexity but achieves very good results in terms of scalability.**

Keywords: Control-based task scheduling, operating systems, multicore processors.

I. Introduction and motivation

Multicore architectures have been widely used as a way to extend computers' performance beyond Moore's law. The multicore chips don't necessarily run as fast as the highest performing single-core models, but they are aimed at improving overall performance by handling more work in parallel [9]. However, exploiting the available parallelism on these platforms is often hindered by many factors, the most important ones being the application parallelism and the operating systems scheduling policies.

Notable and used approaches employ various techniques to adapt or extend existing scheduling functionalities to the multicore case [3]. However, these techniques may result in an increased computational complexity, and in solutions that do not suitably scale with the number of cores. In [4] the authors argue that "high performance on multicore processors requires that schedulers be reinvented". This is a general trend that can be found in all the computing infrastructures [23]. Solutions based on game-theoretical approaches have been proposed [2], [18], [29]. However, they typically focus on specific aspects of interest, such as power consumption minimization or quality of service provisioning. Indeed, the proposed solutions do not adopt a modular and hierarchical approach to the problem, resulting in a lack of flexibility.

Recently, control theory has been applied in the computer science domain with very promising results [10], especially in the context of "self-adapting" or "self-managing" systems [5]–[8], [12], [26]. Initially, the use of control theory consisted in adding another architectural layer to adapt existing systems [21]. Typically, early works gave system components for granted, and added a control layer on top of the existing components to improve their behaviour. Sticking to the application domain of this paper — the operating system scheduling problem — a reservation-based scheduler is controlled by acting on the reservations in order to enhance soft real-time capabilities [1], guarantee quality of service [18], or even guarantee hard real-time constraints [24]. However, a pure control-theoretical design can overcome many of the technological limitations and create "more controllable" systems [11]. Lately, researchers advocated a shift from "controlled" to "control-enabled" computing systems. In control-enabled systems, critical components are entirely designed as controllers [15], [25]. In the following, we take the control-enabled viewpoint and apply it to multicore scheduling.

As discussed in [14], the scheduler of a single-core processor can be realized as a cascade control structure. The inner loop guarantees the desired CPU share among the active threads, while the outer one ensures that the scheduler regains control with a desired period. Comparative tests like those in [17], [19] showed the advantages of the proposed scheduler versus traditional (non control-enabled) ones, in terms of both performance and computational simplicity. In [16] a complete stability proof was provided, and it was also indicated how to translate fairness and responsiveness requirements, as expressed in the Computer Science community, into setpoints for the proposed scheduler. Furthermore, the scheduler was implemented in a microcontroller kernel named Miosix, released under the GPL license[1], and already used for embedded systems [19] and wireless sensor network applications [27], [28].

In this paper, we provide an extension of the mentioned scheduler to the multicore case. In particular, the problem of multicore scheduling is here addressed as a control problem,

A.V. Papadopoulos, Department of Automatic Control, Lund University, Sweden, `alessandro.papadopoulos@control.lth.se`

M. Maggio, Department of Automatic Control, Lund University, Sweden, `martina.maggio@control.lth.se`

A. Leva, Dipartimento di Elettronica, Informazione e Bioingegneria Politecnico di Milano, Italy, `alberto.leva@polimi.it`

R. Carone, graduate student at the Dipartimento di Elettronica, Informazione e Bioingegneria

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.

[1]Miosix is available at `http://www.miosix.org`.

by suitably formalizing the objectives, the control interfaces, and proposing a first set of relocation policies that can take into account different aspects of interest. The aim of this paper is twofold. First, the paper proposes a multicore scheduler, necessary to take advantage of modern architectures and operating systems. Then, it shows how the proposed control approach allows for a neat problem formulation, and leads to solutions that build on top of the developed uniprocessor scheduler, to the advantage of modularity.

II. Background

In this section we provide some background material on control-enabled scheduling in the single-core case. The interested reader can refer to [14], [16] for the details omitted herein.

The scheduling approach of [14] is based on a discrete-time dynamic model for the thread pool that takes the form

$$\begin{cases} \tau_t(k) = \mathbf{b}(k-1) + \delta\mathbf{b}(k-1) \\ \tau_r(k) = \sum_{i=1}^{N} \tau_{t,i}(k) \end{cases} \tag{1}$$

where N is the number of threads, $\tau_{t,i} \in \mathbb{R}^+$, $i = 1,\dots,N$ is the vector of their time allocations, $\tau_r \in \mathbb{R}^+$ is the time between two *rounds*, i.e. two subsequent interventions of the scheduler, counted by the integer index $k \in \mathbb{N}$.

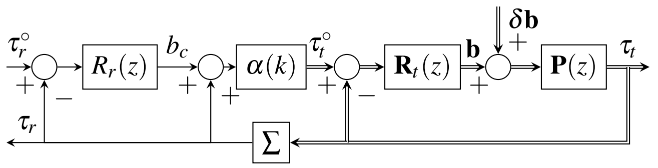

Fig. 1: Block diagram representing single-core scheduling.

Figure 1 shows the scheduler block diagram: $\mathbf{P}(z)$ represents the thread pool, $\mathbf{R}_t(z)$ computes the vector of thread *bursts* $\mathbf{b}(k) \in \mathbb{R}^{+^N}$ to make $\tau_t(k)$ follow a set point $\tau_t^\circ(k)$. The set point is obtained by partitioning the measured $\tau_r(k)$ according to a vector $\alpha(k) \in [0,1]^N$. The vector $\alpha(k)$ is such that $\sum_{i=1}^{N} \alpha_i(k) = 1$. The bursts are additively corrected by the *burst correction* $b_c(k) \in \mathbb{R}$ output by $R_r(z)$, to also have $\tau_r(k) \in \mathbb{R}^+$ follow its set point $\tau_r^\circ(k)$. A diagonal integral regulator is used for $\mathbf{R}_t(z)$, while $R_r(z)$ is a Single-Input Single-Output (SISO) PI.

To generate the vector $\alpha(k)$according to the type of thread, e.g., periodic, sporadic, batch, etc., the approach described in [16] is here used. The proof of stability of the time-varying control system can be found in [16].

Summarising, the single-core scheduler is capable of managing a thread pool maintaining weighed fairness as dictated by $\mathbf{R}_t(z)$, governing responsiveness by means of $\tau_r^\circ(k)$, and facing thread pool variations. On top of this scheduler, that operates at the core level, the multi-core one proposed herein is built.

III. Problem statement and control design guidelines

There are sine main issues in multicore processors, which can be translated into control objectives. Mainly, these issues are balancing the load among different cores and enforcing bounds on the cores heating and consumed power. The second objective is usually reached with techniques like frequency scaling, that reduce the performance of the system, while mitigating the effects of temperature gradients across the chip. Effectively managing how the load is spread among the cores is the key issue to be addressed, and since over-utilized cores are typically the hotspots on the chip, often balancing the load correctly automatically enforces temperature management constraints.

Notice also that, compared to an inter-core context switch, the migration of a thread has a high cost, since it means invalidating the cache lines of the originating core, and a potentially high number of cache misses when the thread is attributed to the destination one. Finally, in some cases load balancing and thermal control may conflict, since the same "load" (as measured with the soft sensors available in modern operating systems) may activate various core units, imposing different power consumption.

For this reasons, in this paper we use as a control objective the "desired load distribution" instead of the more used "load balancing". The former is more general and flexible than the latter. For example, one may want a certain distribution of the loads across the cores for thermal issues, or to deliberately leave a core with a reduced burden in a view to accept load bursts.

Starting from the considerations above, we here sketch the design guidelines.

1) Thread migrations ought to occur with a lower frequency with respect to the control bandwidth of core-level scheduling. This suggests a third loop around the cascade structure of the core-level scheduler of Figure 1.
2) Given the cost of migrations, said third loop is best designed as event-triggered.
3) Modularity and scalability call for a hierarchical structure, where cores internally compute comprehensive indexes signaling their willingness to accept load or their necessity to give some away, while a centralized entity monitors the indexes and enforces the desired policy.

Taking such an approach has several advantages. The first one is a natural isolation with respect to core-level scheduling, a tunable computational effort (for example by using index thresholds for triggering reallocations), a reduced inter-core communication (indexes are computed locally), and the encapsulation of the internal features of cores.

It is worth noticing that state-of-the-art schedulers hardly take into account the issues above. A significant example is the so-called Completely Fair Scheduler (CFS), called SCHED_OTHER in the Linux kernel [13], [22]. The CFS attempts to achieve an even distribution of both CPU share (intra-core) and load (extra-core) by actuating thread relo-

Quantity	Value
Avg. # Migrations per Thread	2098.59
Std. # Migrations per Thread	160.212
Max. # Migrations per Thread	2422
Min. # Migrations per Thread	1689
# Migrations	209859

TABLE I: Statistics on the experiment.

cation with a high frequency (if necessary) and thus a fine time granularity, often resulting in an excessive number of context switches and migration.

In order to show how relevant and time-consuming the absence of a control structure like the one just envisaged could be, we here show some experimental data obtained with a the scheduler benchmarking tool `rt-muse`[2]. The experiments were performed on a machine with an Intel Core i7-3520M (4 cores at 2.9GHz) equipped with the Linux kernel 3.13.0-48 (the most up to date packaged version of the kernel). We run in parallel 100 threads $\{\theta_1, \theta_2, \ldots, \theta_{100}\}$, scheduled with CFS. All the threads can run on 3 of the 4 cores, to avoid invalidating the results by overloading the entire machine. The threads share 11 resources $\{r_0, r_1, \ldots, r_{10}\}$, meaning that they can lock each of the resources in order to perform their tasks. The first resource, r_0, is shared among all the threads, while $\{r_1, r_2, \ldots, r_{10}\}$ are shared by groups of 10 threads each. The second resource, r_1 is shared by the set of threads $\{\theta_1, \ldots, \theta_{10}\}$; r_2 is shared by the set of threads $\{\theta_{11}, \ldots, \theta_{20}\}$, and so on. Each thread θ_i is configured to execute the following operations in loop until stopped.

1) θ_i locks the resource r_0;
2) holding the lock on r_0, θ_i performs 5 thousands mathematical operations;
3) after the operations are terminated, θ_i releases the resource r_0;
4) θ_i performs 10 thousands mathematical operations;
5) θ_i locks the other resource, e.g., r_1 for threads belonging to the set $\{\theta_1, \ldots, \theta_{10}\}$;
6) holding the lock on the second resource, θ_i performs 5 thousands mathematical operations;
7) θ_i releases the locked resource.

The threads continuously execute the loop until stopped. In our experiment, we stopped the threads after 25 minutes. During the 25 minutes, we logged all the occurred migrations, including the time a migration happened, the thread that migrated, and source and destination for the migration. During the experiments threads were migrated 209859 times. Table I shows some statistics on the experiment. Notice that even the thread that migrates the least, still migrates about 1.126 times per second. Figure 2 shows the distributions of the frequencies of the inter-arrival time between two subsequent migrations of a single thread, for each core. It is worth noticing that most of the migrations happen with an inter-arrival time that is less than 100ms, while some of

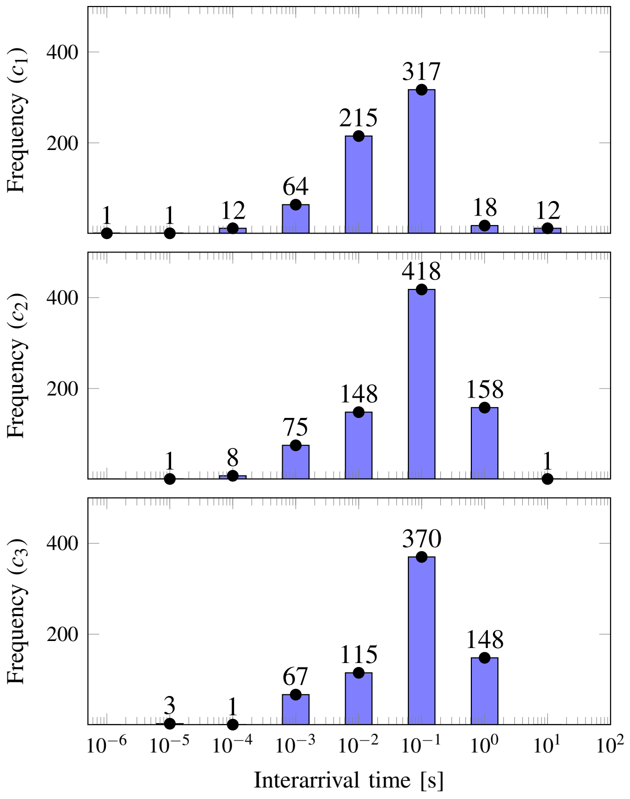

Fig. 2: Frequencies of the interarrival time for each core with CFS.

them happen with an inter-arrival time that is even in the time scale of some micro-seconds, i.e., even lower than the typical thread execution time (the interval of time between when the thread starts executing and when the scheduler regain control over the core).

Summarizing, the authors believe that present multicore schedulers lack a modular structuring, and that the consequent actuation policies can result in much time wasted in useless thread relocations, and even fall short of perfection at achieving load, thermal, and power balance. The solutions proposed herein attempt to address such issues.

IV. The proposed control scheme

The proposed scheme has a hierarchical structure. Each core hosts a scheduler, endowed with an additional software component called the *core load monitor*. The purpose of the monitor is to compute an index indicating whether the core needs to have its load reduced, or is on the contrary willing to accept new threads. All the cores communicate with another component, a central *thread dispatcher*, which can force the migration of a thread from a core to another. In our scheme, the thread dispatcher runs on the first core that boots and awakens the others. The dispatcher has access to the indexes computed by the monitors via the processor internal bus, and uses the provided information to take decisions.

[2]`rt-muse` is publicly available at `https://github.com/martinamaggio/rt-muse`

A. Preliminaries and notation

Definition 1 (Multicore system): A *multicore system* can be characterized by the following quantities.

- $\mathbb{C} = \{c_1, c_2, \dots, c_{N_c}\}$ is the set of N_c cores in the system. Each core executes a scheduler like that of Figure 1.
- $\mathbb{T} = \{\theta_1, \theta_2, \dots, \theta_{N_\theta}\}$ is the set of N_θ threads running in the system.
- $S : \mathbb{N} \times \mathbb{T} \times \mathbb{C} \to \{0, 1\}$ is a *localization function* which determines at each time instant k, what if a thread $\theta \in \mathbb{T}$ is running on the core $c \in \mathbb{C}$. Therefore, $S(k, \theta, c) = 1$ if θ is running on c, $S(k, \theta, c) = 0$ otherwise.
- $\alpha_\theta(k) \in [0, 1]$ for $\theta \in \mathbb{T}$ is the CPU share that the thread θ desires. The values of $\alpha_\theta(k)$ are such that

$$\sum_{\theta \in \mathbb{T}} \alpha_\theta(k) \leq N_c \qquad \forall k \in \mathbb{N}.$$

Notice that each $\alpha_\theta(k)$ is in the range $[0, 1]$ because a single thread cannot occupy more than the totality of the CPU time provided by one core. Consistently, the sum of all the $\alpha_\theta(k)$ is supposed to not exceed N_c. Recall that, for each core, $\alpha_\theta(k)$ represents the percentage of the core that should be used by thread θ at time k. Here, some CPU time may be *deliberately* left idle, thereby allowing to purposely under-utilize a core if this is necessary. To recover the case of [14] one can simply suppose that convenient "slack" distribution coefficients are introduced, but this is neglected in the following as it is irrelevant for the aim of this work.

In the following, to lighten the notation, we shall treat all the quantities just mentioned as available processor-wide, although in a real implementation they are composed of elements updated by different entities in different instants.

B. The core load monitors

At each k-th intervention of its local scheduler, the load monitor of core $c \in \mathbb{C}$ can compute the *local load request* $L_c(k)$ of that core as

$$L_c(k) = \sum_{\theta \in \mathbb{T}} S(k, \theta, c) \alpha_\theta(k) \qquad \forall c \in \mathbb{C}. \tag{2}$$

Now, suppose that the core is requested by the upper levels of the control hierarchy to have an actual load $L_c^\circ(k)$. Therefore, one can define the core *normalised load error* as

$$\varepsilon_c(k) = \frac{L_c(k) - L_c^\circ(k)}{L_c(k)} \qquad \forall c \in \mathbb{C}, \tag{3}$$

where $\varepsilon_c > 0$ means that the core would need unloading, while $\varepsilon_c < 0$ signals that it could accept additional threads.

Finally, the quantities made available processor-wide by each load monitor are $\varepsilon_c(k)$ as per (3), and the *time integrated overload index* $O_c(k)$. $O_c(k)$ is reset every time a thread is moved from core c to another core. Defining k_c^0 as the first intervention of the scheduler of core c following the last thread reallocation performed by the dispatcher on threads that were executing on c, the overload index $O_c(k)$ becomes

$$O_c(k) = \sum_{h=k_c^0}^{k} \max\left(0, \varepsilon_c(h)\right) \tau_{rc}(h) \qquad \forall c \in \mathbb{C}, \tag{4}$$

where $\tau_{rc}(\cdot)$ is the round duration of core c.

C. The thread dispatcher

The dispatcher runs contextually with the scheduler of the hosting core 1, and since it is expected that its actions are sporadic with respect to core-level scheduling, the round duration set point for the hosting core is largely irrelevant. The thread dispatcher operation can be summarized as follows.

1) The dispatcher reads the desired core loads;
2) it then read the last values of O_c and ε_c made available by each core;
3) based on the received information, it decides whether or not to perform a reallocation. The simplest *reallocation triggering rule* is to check that at least one O_c exceeds a threshold $\overline{O}_c$ and at least one ε_c is negative, i.e., at least one core is not over-utilized;
4) if the dispatcher wants to perform a migration, the following steps are executed:
    a) the dispatcher determines the source $s \in \mathbb{C}$ and destination $d \in \mathbb{C}$ cores, and the thread(s) to move, with a convenient *dispatching rule*;
    b) it then uses the available operating system primitives to move the thread(s);
    c) finally, it resets the overload index $O_s(k)$ of the source core s to zero, and k_0 of (4) to the current time.
5) if no migrations have to be performed, nothing happens and the core is released for the execution of other activities.

One of the advantages of the adopted triggering rule is the possibility to enforce a modular and hierarchical control structure. For example, if some cores are over-utilized for long time but none can accept load, this could be forwarded to some other higher-level controller that can act on voltage and frequency to compensate for it, if necessary. The proposed scheme makes such controllers well identified, with precisely defined inputs and outputs, and could be composed in convenient (e.g., override) structures to deal with conflicting requirements in a formally specified manner. The choice of the threshold $\overline{O}_c$ how fast the system is to detect an overload. Care has to be taken for its choice, since too low values of the threshold usually result in a higher number of migrations, while too high values may result in no migrations but temperature issues since one core is continuously overloaded.

As can be seen, the communication between the thread dispatcher ant the core load monitors has an asynchronous nature, where the data made available by the latter entities to the former play the role of the communication shared memory. This can clearly cause some delay in the intervention of the dispatcher. However, that delay is structurally limited to twice the maximum τ_{rc} among the cores. Thanks to the underlying control-based scheduler, a reliable estimate of that delay is thus twice the maximum *set point* for the round times. The delay under question could be reduced by conveniently managing `runqueue`-like data structures at the core level, which is beyond the scope of this work. In any case, if sporadic reallocations happen in the system, this becomes of low importance.

V. Proposed dispatching rules

A migration at time k is uniquely determined by three components $\mathscr{M}_k = (s, \theta, d)$. The source $s \in \mathbb{C}$ is determined by the load monitor whenever $O_s > \overline{O}_s$, and the set of the threads running on s is defined as $\mathbb{T}_s := \{\theta \in \mathbb{T} | S(k, \theta, s) = 1\}$. The thread $\theta \in \mathbb{T}_s$ to be migrated, and the destination core $d \in \mathbb{C} \setminus \{s\}$ must be determined by the dispatching rule. We here propose four different dispatching rules that can be adopted for thread migration in a multicore scheduler. The advantage of these policies relies on their low computational complexity, allowing for their use in a real-time system.

A. Simple

In this dispatching rule a random thread θ is selected among those running on the core(s) that signaled the necessity of a relocation. The destination source is selected as

$$d = \arg\min_{c \in \mathbb{C} \setminus \{s\}} L_c(k) \tag{5}$$

The main advantage of such a rule is computational simplicity.

B. Load aware

Let us define the spare capacity of a core $c \in \mathbb{C}$ as

$$a_c(k) = (L_c^\circ - L_c(k))^+ \qquad \forall c \in \mathbb{C}, \tag{6}$$

where the x^+ represents the positive part of the number x. The thread $\theta \in \mathbb{T}_s$ to be migrated, and the destination core $d \in \mathbb{C} \setminus \{s\}$ can be selected as

$$(\theta, d) = \arg\max_{i \in \mathbb{T}_s, c \in \mathbb{C} \setminus \{s\}} (a_c(k) - \alpha_i(k)). \tag{7}$$

The idea of this policy is to migrate threads with low CPU share request to the core with the highest spare capacity.

C. Load normalised

In the two previous policies, the desired utilization L_c° of each core is just an input to the system, and it is constant. This policy is a natural extension of Load aware. The idea is to adapt the value of the desired utilization every Δ_U interventions as

$$L_c^\circ(k) = \frac{1}{N_c} \sum_{c \in \mathbb{C}} L_c(k), \qquad \forall c \in \mathbb{C}, k = \Delta_U, 2\Delta_U, \ldots \tag{8}$$

namely, the desired load is obtained as the average of the actual loads. The underlying idea of this policy is to balance the overall load among the different cores by dynamically adapting the set points. Then, the Load aware rule is applied for the relocation as above. In this case, the adaptation period Δ_U is the only parameter of the method.

Notice that this policy could be easily modified with a weighted distribution of the load among the cores, for instance introducing weights dependent on the temperature $T_c \in \mathbb{R}$ of the core as

$$L_c^\circ(k) = w_c(k, T_c) \sum_{c \in \mathbb{C}} L_c(k), \quad \forall c \in \mathbb{C}, k = \Delta_U, 2\Delta_U, \ldots \tag{9}$$

$$\text{with} \sum_{c \in \mathbb{C}} w_c(k, T_c) = 1, \quad w_c(k, T_c) \in [0, 1]. \tag{10}$$

In this case, w_c can be chosen close to zero if the temperature of the core is high, and close to one otherwise.

D. Turn-over

This solution is analogous with an old but quite assessed management policy for overload prevention, named "stop and go" [20] since one or more cores are stopped when necessary, typically by means of clock gating. The idea here is to extend stop and go by (a) allowing a core to have its available computational power reduced, without however being disabled completely, and (b) determine which core(s) to temporarily "unload" based on the proposed indexes. Also, here the load of the unloaded core(s) is relocated to the others, which reduces the service quality degradation, and further motivates the different name of "turn over". The parameters of this policy, in addition to $\overline{O}_c$, are

- the off-time Δ_{TO} for the unloaded core(s),
- and $\underline{L}^\circ$: the desired utilization setpoint when a core is in an off-time phase (zero if it is disabled completely).

The policy is actuated by unloading the core $c_{\text{off}} \in \mathbb{C}$ that signals the necessity of a thread reallocation by moving the core setpoint to $\underline{L}^\circ$, while the other setpoints can be computed as

$$L_c^\circ(k) = \frac{1}{N_c - 1} \sum_{c \in \mathbb{C}} L_c(k), \tag{11}$$

and then applying the Load aware rule above. Notice that the main purpose of this approach is not meant to minimize the number of migration, but to spread the load among the cores with low power consumption, while unloading the ones with high power consumption.

VI. Simulation examples

We here present the results obtained with the proposed policies in two different scenarios. The results have been performed using a python simulator[3]. For all the experiments the load monitor has $O_c = 2$, $\forall c \in \mathbb{C}$.

A. First scenario: 32 threads on 4 cores

In the first scenario the thread pool is composed of 32 threads, that must be run on 4 cores, for 200 scheduling rounds. The $\alpha_\theta(k)$ is constant and equal for all the considered threads. The round time τ_{rc} is equal for all the cores. In the experimental results, we omit the Load aware policy, since the figures are qualitatively identical to the ones obtained with the simple one.

Figure 3 shows the results obtained with the *Simple* dispatching rule. The plots on the left column show the

[3]The simulator is released under the GPL license and publicly available at `https://github.com/apapadopoulos/MultiCoreMigrationSimulator`

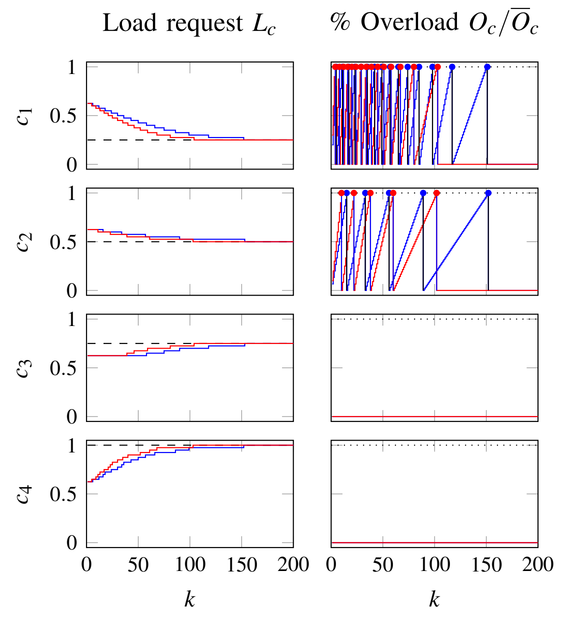

Fig. 3: Simulation results with the *Simple* dispatching rule (22 migrations with $O_c = 2$, 20 migration with $O_c = 3$).

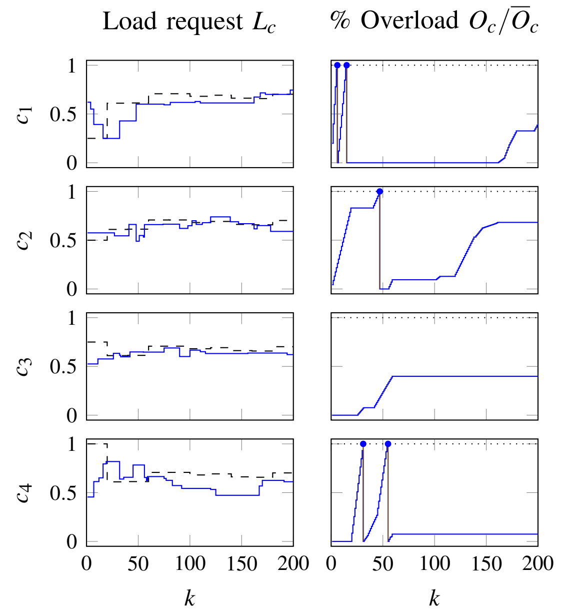

Fig. 4: Simulation results with the *Load normalised* dispatching rule (5 migrations).

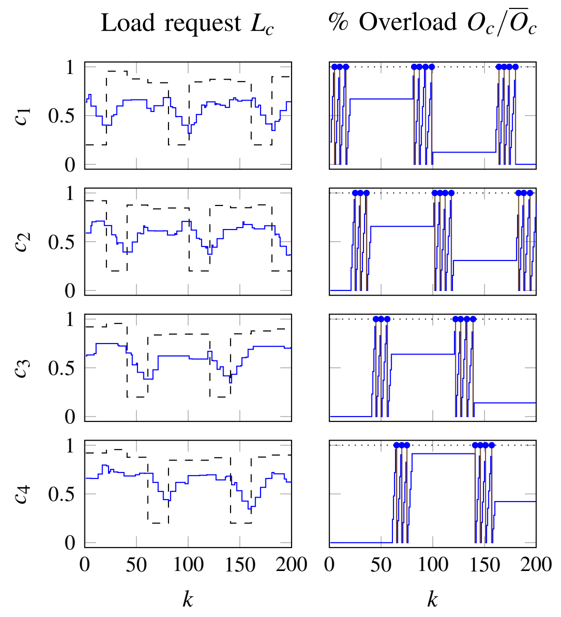

Fig. 5: Simulation results with the *Turn-over* dispatching rule (35 migrations).

requested load while the plots on the right column show the percentage of overload, computed as $O_c/\overline{O}_c$. The utilization setpoint for the four cores are $L^{\circ}_{c_1} = 0.25$, $L^{\circ}_{c_2} = 0.5$, $L^{\circ}_{c_3} = 0.75$, and $L^{\circ}_{c_4} = 1.0$ respectively. The blue line refers to a simulation with with the threshold $\overline{O}_c$ is set to 3, while the red line shows the result for $\overline{O}_c$ set to 2. As can be observed, changing the threshold speeds up the convergence of the migrations. The rule migrates threads from the cores that are overloaded with respect to their setpoint (in the example, c_1 and c_2), to the ones that are not overloaded. More precisely, when the integrated overload index reaches the threshold, a thread is migrated out of the corresponding core and the index is reset. If the overload index reaches the threshold again, this causes a subsequent migration.

The results of the *Load normalised* dispatching rule are shown in Figure 4. We set $\Delta_U = 20$ so as to adapt the utilization setpoint every 20 rounds. The initial utilization setpoint is the same as for the simple policy, and it is adapted according to (8). The convergence to the set point is slower, but on the other hand the number of migrations is lower than with the *Simple* rule.

Figure 5 shows the results obtained with to the *Turn-over* dispatching rule. In this case, intuitively, the core utilization set points take essentially the role of an upper bound on their burden, as can be seen in the results. The number of migrations can be high, especially for the re-distribution of the unloaded cores' work, but migrations tend to be clustered in specific small time intervals, while for the rest of the time no thread is migrated. The parameters chosen for this example are $\Delta_{TO} = 20$, $SB = 0.2$ and $\overline{O}_c = 2$.

B. Second scenario: Scalability

In the second scenario we analyze how the number of migrations scales with the number of cores and with the number of threads. Since the turn-over policy is not meant to minimize the number of migrations, but it is based on Load aware, we here do not consider it. We simulated 1500 scheduling rounds, with the a varying number of threads, $N_\theta \in \{100, 128, 250, 256, 500, 512, 1000, 1024\}$, and with a varying number of cores, $N_c \in \{2, 4, 8, 16, 32, 64, 128, 256\}$. At the beginning of the simulation, the threads are all on the first core, while all the other cores are unloaded, which is the most critical situation.

Figure 6 shows the obtained results as a function of the

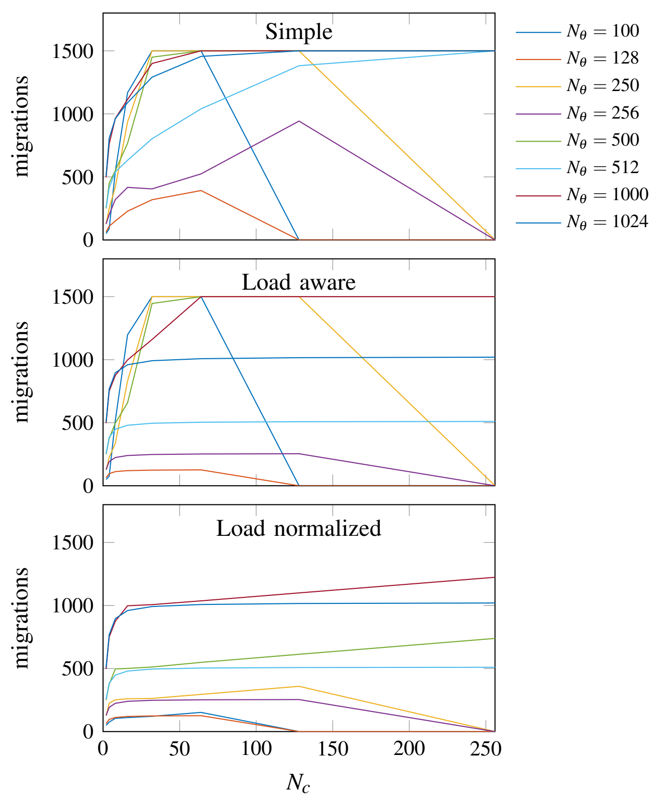

Fig. 6: Number of migrations as a function of the number of cores.

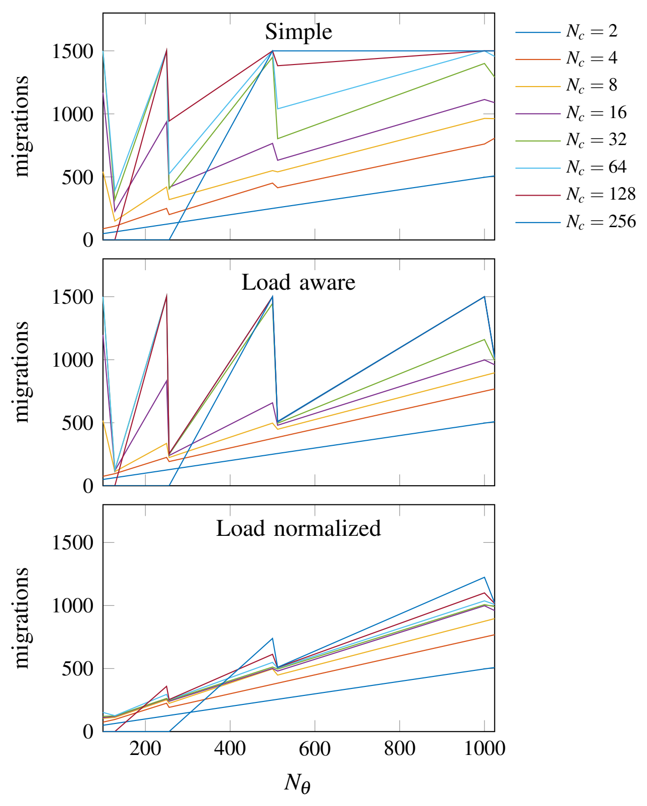

Fig. 7: Number of migrations as a function of the number of threads.

number of cores, while Figure 7 presents the same information as a function of the number of threads. The number of migrations in all the policies has a linear relationship with respect to the number of threads while it seems to be fairly constant with respect to the number of cores. This is an interesting result. Indeed, a migration policy should not be affected by the number of cores.

In the end one can conclude that Load normalization behaves better than the other policies both in the first and in the second scenario. Indeed, Load normalization allows to obtain the least amount of migrations, and a more predictable behaviour. At the same time its computational complexity is limited.

VII. Conclusions and future work

We have revisited the problem of multicore thread scheduling and migration with a control-theoretical attitude, extending previous uniprocessor results. We have formulated the problem as a control problem, allowing for the introduction of suitable interfaces and objectives that can be achieved with a modular and hierarchical control structure. The contribution of this paper is the design of the thread dispatcher. The dispatcher works on top of the uniprocessor scheduler and serves as a higher-level controller, possibly implementing a simple policy as well as a power-aware one. We have also proposed some potential policies, that could be further refined. We believe that the main contribution of this paper resides in the structure of the overall control mechanism. The proposed structure allows for the design and implementation of novel techniques that are able to easily scale up to the multicore case, while maintaining a low computational complexity. We have shown with simulation results the behaviour of the policies and developed a simulator that is distributed freely to foster future research in multicore scheduling.

References

[1] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, “Analysis of a reservation-based feedback scheduler”, in *Proc. 23rd IEEE Real-time Systems Symposium (RTSS02)*, Austin TX, USA, 2002, pp. 71–80.

[2] I. Ahmad, S. Ranka, and S. Khan, “Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy”, in *Parallel and Distributed Processing. IEEE Int. Symposium on*, 2008, pp. 1–6.

[3] J. Anderson, J. Calandrino, and U. Devi, “Real-time scheduling on multicore platforms”, in *Real-Time and Embedded Technology and Applications Symposium. Proc. of the 12th IEEE*, 2006, pp. 179–190.

[4] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, “Reinventing scheduling for multicore systems”, in *Proc. of the 12th Conf. on Hot Topics in Operating Systems*, ser. HotOS’09, Berkeley, CA, USA: USENIX Association, 2009, pp. 21–25.

[5] S. Chauhan, A. Sharma, and P. Grover, “Developing self managing software systems using agile modeling”, *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 6, pp. 1–3, 2013.

[6] S. Cheng, V. Poladian, D. Garlan, and B. Schmerl, “Engineering self-adaptive systems through feedback loops”, in *Software engineering for self-adaptive systems*, B. Cheng *et al.*, Eds., Berlin, Germany: Springer, 2009, pp. 48–70.

[7] Y. Diao *et al.*, “Self-managing systems: A control theory foundation”, in *Proc. 12th IEEE Int. Conf. and Workshops on Engineering of Computer-Based Systems*, Phoenix, AZ, USA, 2005, pp. 441–448.

[8] A. Filieri *et al.*, “Software engineering meets control theory”, in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 15, 2015.

[9] D. Geer, "Chip makers turn to multicore processors", *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[10] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback Control of Computing Systems*. Hoboken, NJ, USA: John Wiley & Sons, 2004.

[11] C. Karamanolis, M. Karlsson, and X. Zhu, "Designing controllable computer systems", in *Proc. 10th Conf. on Hot Topics in Operating Systems*, Berkeley, CA, USA, 2005, pp. 9–15.

[12] J. Kephart and D. Chess, "The vision of autonomic computing", *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[13] A. Kumar, "Multiprocessing with the completely fair scheduler", *IBM developerWorks*, 2008.

[14] A. Leva and M. Maggio, "Feedback process scheduling with simple discrete-time control structures", *Control Theory Applications, IET*, vol. 4, no. 11, pp. 2331–2342, 2010.

[15] A. Leva, M. Maggio, A. Papadopoulos, and F. Terraneo, *Control-based Operating System Design*. London, UK: IET, 2013.

[16] A. Leva, A. Papadopoulos, and M. Maggio, "A general control-theoretical methodology for runtime resource allocation in computing systems", in *Decision and Control (CDC), 2013 IEEE 52nd Annual Conf. on*, 2013, pp. 3487–3492.

[17] M. Maggio, F. Terraneo, A. Papadopoulos, and A. Leva, "A PI-based control structure as an operating system scheduler", in *Proc. IFAC Conf. on Advances in PID Control PID'12*, 2012, pp. 329–334.

[18] M. Maggio, E. Bini, G. C. Chasparis, and K.-E. Årzén, "A game-theoretic resource manager for RT applications", in *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, 2013, pp. 57–66.

[19] M. Maggio, F. Terraneo, and A. Leva, "Task scheduling: A control-theoretical viewpoint for a general and flexible solution", *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, 76:1–76:22, 2014.

[20] R. Mall, *Real-Time Systems: Theory and Practice*. Pearson Education, 2009.

[21] P. Oreizy *et al.*, "An architecture-based approach to self-adaptive software", *IEEE Intelligent systems*, vol. 14, no. 3, pp. 54–62, 1999.

[22] C. Pabla, "Completely fair scheduler", *Linux Journal*, vol. 2009, no. 184, Article No. 4, 2009.

[23] A. V. Papadopoulos, "Design and performance guarantees in cloud computing: Challenges and opportunities", in *10th International Workshop on Feedback Computing*, 2015.

[24] A. V. Papadopoulos, M. Maggio, A. Leva, and E. Bini, "Hard real-time guarantees in feedback-based resource reservations", *Real-Time Systems*, vol. 51, no. 3, pp. 221–246, 2015.

[25] A. V. Papadopoulos, M. Maggio, F. Terraneo, and A. Leva, "A dynamic modelling framework for control-based computing system design", *Mathematical and Computer Modelling of Dynamical Systems*, pp. 1–21, 2014.

[26] M. Salehie and L. L. Tahvildari, "Self-adaptive software: Landscape and research challenges", *ACM Tran. on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, pp. 1–42, 2009.

[27] F. Terraneo *et al.*, "FLOPSYNC-2: Efficient monotonic clock synchronisation", in *Proceedings of the 35th IEEE Real-Time Systems Symposium*, ser. RTSS, 2014, pp. 11–20.

[28] F. Terraneo *et al.*, "Reverse flooding: Exploiting radio interference for efficient propagation delay compensation in wsn clock synchronization", in *Proceedings of the 36th IEEE Real-Time Systems Symposium*, ser. RTSS, 2015.

[29] G. Wu, Z. Xu, Q. Xia, and J. Ren, "An energy-aware multi-core scheduler based on generalized tit-for-tat cooperative game", *Journal of Computers*, vol. 7, no. 1, 2012.