Analysis and experimental validation of processor load for event-driven controllers[1]

J.H. Sandee[§], P.M. Visser[†] and W.P.M.H. Heemels[‡]

[§]Technische Universiteit Eindhoven
Dept. of Electrical Engineering, Control Systems Group
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: j.h.sandee@tue.nl

[†]University of Twente, Dept. of Electrical Engineering, Email: p.m.visser@utwente.nl
[‡]Technische Universiteit Eindhoven, Dept. of Mechanical Engineering, Email: m.heemels@tue.nl

***Abstract*— Event-driven controllers differ from the standard digital controllers as their sample times are generally not periodic (time equidistant). In literature several proposals for event-driven controllers are made in order to reduce the number of control updates and consequently the processor load needed for its implementation. This is possible without deteriorating the control performance significantly. However, experimental validation has not been presented in literature. This paper aims at filling this gap. Simulations, as well as experiments on a copier paper path test setup, show that a reduction in the number of control updates indeed results in a considerable reduction of the processor load, with only a small decrease of control performance. Furthermore, we present a method to predict the processor load very accurately, without having to implement the controller on a test setup.**

I. Introduction

In most applications nowadays, digital controllers are implemented on embedded hardware with strong requirements for both the control performance as well as the processor load. Often, a high update rate of the controller algorithm is chosen to be able to guarantee good control performance. This, however, evokes high processor loads. Conventionally, designers try to reduce the sample frequency of the digital controller as much as possible, to minimize the processor load, while keeping in mind the (minimal) required control performance. In almost all of these designs, the sample frequency is taken constant, creating a constant distribution of the processor load for the specific control task.

The sample frequency is normally chosen on the requirement to track fast changing reference signals or to reject high bandwidth disturbance. However, in many cases reference signals are not changing continuously and severe disturbances appear only sporadically. Only during these periods a high sample frequency is needed, while in other periods of time we do not have to require the same (high) sample frequency of the controller. This rationale indicates that it would be beneficial to vary the sample frequency to optimize over both the control performance and the processor load at the same time.

In literature [1], [2], [4], [5], [6], event-driven (or asynchronous) controllers are proposed to make this trade-off between control performance and processor load. Specially designed event-generating mechanisms take care of triggering the controller to update the actuator signal at specified moments in time. Henriksson et al. [4] use for example optimal controllers to distribute processing power between three controllers running at varying sample rates. Årzén [1] uses an event-based strategy in combination with a standard PID-controller in which the sample frequency of the controller is chosen relative to the derivative of the measured tracking error. In [2] a similar structure was proposed to reduce the number of actuator updates. In [5] and [6] various control structures are analyzed in which the sample frequency is chosen relative to the absolute value of the measured tracking error. When the error is small, fewer or even no computations are carried out and the actuator signal is held constant. Therefore, the controller will then put no effort in making the error even smaller, reducing the processor load at those periods of time.

In the above mentioned literature, all indications of processor load reduction are obtained by simulating or analyzing only the update rate of the controller. Several assumptions are made to relate the simulated number of control updates to the processor load, but experimental evidence has not been presented in literature so far. One common assumption is that only relatively few overhead is needed to implement the event-generating mechanism of the event-driven controllers. Furthermore, it is assumed that the execution of a time synchronous task takes the same time as the execution of an asynchronous task with the same average rate of occurrence. This does not necessarily have to be the case, due to for instance cache memory. For these reasons, the event-driven controller could even reduce the number of control updates, while increasing the processor load.

[1]This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

[‡]Maurice Heemels is also at the Embedded Systems Institute, Eindhoven, The Netherlands.

The purpose of this research is to validate the promise of event-driven controllers to reduce processor load. Moreover, this research investigates the relation between the reduced number of control updates and the processor load. Both a time-driven as well as an event-driven controller are implemented on an experimental setup of a copier paper path, driven by DC-motors. Measurement data will be compared with simulation data to validate the assumptions made in [5] and [6]. Furthermore, we will investigate the possibility to predict the processor load beforehand, without having to implement the controller on a test setup.

II. Event-driven controller

We consider the plant described by

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ y(t) &= h(x(t)) \end{aligned} \tag{1}$$

where $x(t) \in \mathbb{R}^n$ is the state, $u(t) \in \mathbb{R}$ the control input and $y(t) \in \mathbb{R}$ the output, respectively, at time $t \in \mathbb{R}_+$. $f : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ and $h : \mathbb{R}^n \to \mathbb{R}$ can be linear as well as non-linear functions.

In conventional digital control a fixed sample time T_s is used, meaning that the *event times* are equal to kT_s. We call this a time-driven controller. In this paper we use a digital PI feedback controller. The chosen controller is given by the following difference equation:

$$u_k = Pe_k + (IT_s - P)e_{k-1} + u_{k-1}, \tag{2}$$

where $e_k := y(kT_s) - r_k$ is the tracking error at time $t = kT_s$ and $r_k = r(kT_s) \in \mathbb{R}$ is the value of the reference signal at time $t = kT_s$. By using the zero-order hold $u(t) = u_k$ for all $t \in [kT_s, (k+1)T_s)$. P and I are the proportional and integral gain of the PI controller.

To reduce the number of required control calculations and actuator signal updates, we propose to not update the control value if the error e_k is smaller than a threshold value e_T. If the error is smaller than e_T at $t = kT_s$, the controller output will not be calculated and updated. If the error is larger than e_T, an update is performed according to (2).

Hence, the controller (2) is modified to

$$u_k = \begin{cases} Pe_k + (IT_s - P)e_{k-1} + u_{k-1} & \text{if } |e_k| > e_T \\ u_{k-1} & \text{if } |e_k| \leq e_T \end{cases} \tag{3}$$

where again $u(t) = u_k$ for all $t \in [kT_s, (k+1)T_s)$. Note that whether the error is smaller than e_T, is still detected on a constant rate (at times $t = kT_s$).

The aim of the control design (selecting T_s, e_T, P and I) is to get good control performance (in the sense that the maximal tracking error $e_{max} := \max_{t \in \mathbb{R}_+} |y(t) - r(t)|$ is acceptable) and the processor load is small.

III. Experimental setup

A. Plant

Figure 1 depicts a photograph of the experimental setup. It represents the paper path of a copier that consists of 5 identical motors which drive 5 pinches. These pinches drive the sheets of paper through the paper path, from the paper input module (PIM), to the output tray.

For this case study we consider only one motor of the paper path, which is a Maxon RE25 20 Watt motor. Its axis is coupled to the pinch with a stiff belt. To the other end of the motor axis, a 500-slit rotary encoder is connected. This signal is acquired with quadrature demodulation, resulting in a resolution of 2000 counts per rotation. An H-bridge amplifier is used to control the motor. It operates at 22 volts and is limited to a maximum current of 3 amps.

B. Model

The model that is used for simulation and synthesis of the controller was built in 20-sim (a simulation package developed at the University of Twente [7]). The model consists of an accurate description of the motor (delivered by the Maxon Motor company), a model of the load together with a non-linear friction model of the bearings, the PI controller and the quantizing effects caused by the encoder and H-bridge amplifier. In terms of equation (1) $y(t)$ is the angular velocity of the motor and $u(t)$ is the motor voltage.

C. Control system

The control system consists of a PC104+ CPU board with a 600 MHz x86 compatible CPU, supplied with 256 MB RAM and a 32 MB Flash disk which contains the RTAI operating system. An FPGA is connected to the CPU board via the PCI bus in order to perform the I/O operations. In this setup the configuration for the FPGA contains a pulse width modulated (PWM) output and encoder quadrature input. The PWM output signal that drives the H-bridge amplifier, has a frequency of 16kHz. The duty-cycle of the PWM signal can be set in 2048 steps. A separate signal output determines the rotation direction of the motor. The CPU sets the duty-cycle and direction and the FPGA keeps these values until a new value has been received (implementing the $u_k = u_{k-1}$, equation 3). The encoder input increments a counter at the FPGA on every pulse received from the encoder. The CPU can read this counter.

D. Controller design

To control the angular velocity of the motor, the time-driven PI controller as given in (2) was tuned using common design rules and implemented on the test setup. The resulting

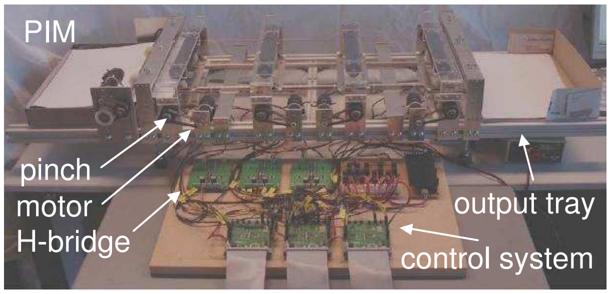

Fig. 1. Photograph of the paper path setup.

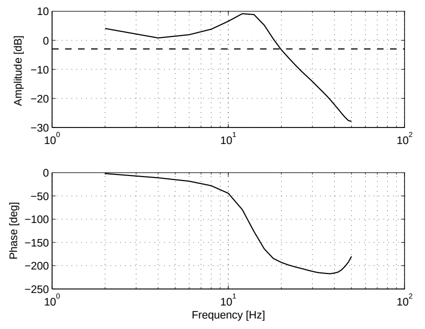

Fig. 2. Closed-loop response of the motor.

closed-loop response (from reference velocity r to output velocity y), depicted in figure 2, was derived from the frequency response of the sensitivity function. The sensitivity function was experimentally determined at the setup, by injecting white noise at the actuator signal and measuring the control output signal u. Note that the values for frequencies below 6 Hz should be considered uncertain, as the coherence of the sensitivity measurement was far below 1 for these frequencies. In the figure, the bandwidth is indicated at approximately 20 Hz. Rules of thumb advise to set the controller frequency at a minimum of 6 times the closed-loop bandwidth (see e.g. chapter 11 in [3]). We chose the sample frequency at 100 Hz, which is only 5 times the obtained closed-loop bandwidth. The choice for this low sample frequency was made to assure an initial low processor load for both the time-driven and the event-driven controller.

The event-driven controller, as given in (3), was implemented using the same sample frequency of 100 Hz. The event-driven controller can be written in pseudo code as follows:

```
1  pos = input(encoder);
2  vel = (pos - previous(pos))/Ts;
3  error = reference - vel;
4
5  if (error > eT OR error < -eT) then
6    uP = P*error;
7    uI = previous(uI)+I*previous(error)*Ts;
8
9    u = limit(uP+uI, min_u, max_u);
10
11   motor_voltage = output(u);
12 end;
```

The time-driven controller was implemented in a similar way, by omitting the lines with numbers 5 and 12. Note that this indicates clearly the overhead introduced by the event-driven controller. The benefit for the event-driven controller can also be observed, as lines 6 to 11 are only carried out under specific conditions.

IV. Simulation results

The simulation results for the time-driven controller are depicted in figure 3. The first graph gives the velocity reference signal in rotations per second (rot/s). In the setup, this profile can be used for every motor (but shifted over time) to drive several sheets of paper through the paper path

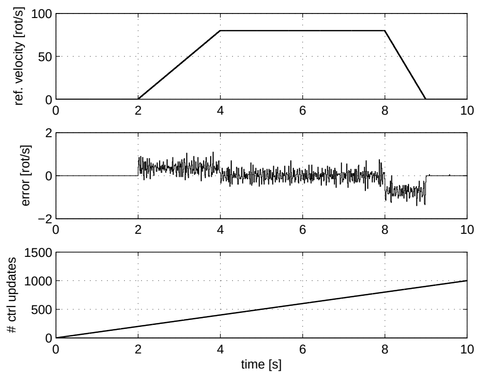

Fig. 3. Simulation results of time-driven controller.

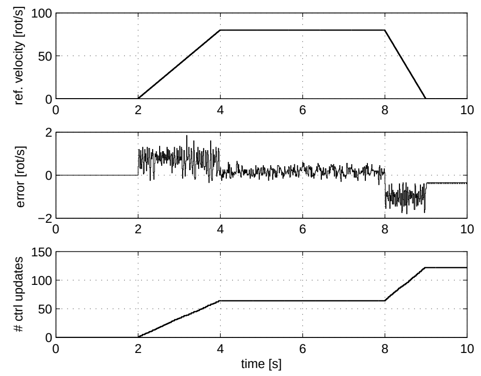

Fig. 4. Simulation results of event-driven controller for $e_T = 0.9$ rot/s.

sequentially. The profile starts and ends with a period of zero velocity. During this period, no sheets need to be transported and therefore the motor can halt. The second graph shows the error of the controller. When the motor is running at non-zero velocity, the error demonstrates a noisy behavior. This is caused by the belt that inserts relatively high frequent disturbances in the system. Although hardly visible, the error signal is quantized in steps of 0.05 rot/s (100 samples/s over 2000 counts/rot). The third graph shows the number of control updates for the time-driven controller, which is of course linearly increasing in time with 100 updates per second for the 100 Hz controller.

The same signals are plotted in figure 4 for the event-driven controller simulation with $e_T = 0.9$ rot/s. The reference velocity is chosen the same, like in all other simulations and experiments in this paper. As expected, the second plot shows a larger error (up to 1.85 rot/s) compared to the time-driven controller simulation (error up to 1.2 rot/s). When the motor is running at constant velocity, the error stays below the specified bound e_T. The third graph shows how the number of control updates increases over the simulation. It can clearly be seen that when the motor is in stand-still, or when the motor is running at constant velocity, no control updates are needed. However, when more severe disturbances are present updates might also be necessary in the constant velocity phases. An example of such a disturbance could be a sheet of paper that is traveling through the paper path, creating a disturbance torque onto the motor. In the third

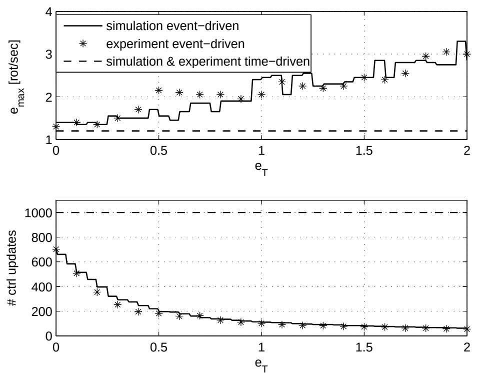

Fig. 5. Experimental and simulation results of time-driven and event-driven controller. The event-driven controller was implemented for various values of e_T.

graph it can also be seen that the controller takes more control updates per second when decelerating compared to accelerating. This can be explained by the fact that the controller is followed by a zero-order hold and that the motor decelerates faster than it is accelerating. Therefore, more actuator signal updates are needed to follow the varying velocity with comparable performance.

As the control performance measure we use the maximum error (e_{max}) over each 10 seconds simulation, as described in section II. When we increase e_T, the maximum error increases. This relation between e_T and e_{max} is investigated and depicted in the first plot of figure 5 (solid line). In this plot, the simulation results for 300 different values of e_T in the range [0, 2] are given. The straight dashed line in this figure visualizes e_{max} for the time-driven controller simulation (being 1.2 rot/s). This value is somewhat lower than the lowest maximum error that the event-driven controller can achieve, which is realized for $e_T < 0.25$ rot/s. This value of 1.3 rot/s is also the offset for the linearly increasing trend in e_{max} that can be observed for increasing values of e_T for the event-driven controller. Like the error signal, the value of e_{max} is quantized with a minimal step size of 0.05 rot/s.

The solid line in the lower graph of figure 5 shows how e_T relates to the total number of control updates over the 10 seconds simulation for the simulation results. When in the event-driven controller e_T is set to 0 rot/s, the total number of control updates already decreases from 1,000 to 700. This is because the reference velocity is zero for 3 seconds. No motor voltage needs to be applied to keep the motor in stand-still. This is only true when no excessive disturbance is present that could force the angular velocity of the motor to a non-zero value. The quantization effect can again be observed, as the total number of control updates only changes at values of e_T at a multiple of 0.05 rot/s.

V. Prediction

From the number of control updates, obtained from the simulations, we can predict the processor load. For this, we need the processing time needed to execute the particular actions in controller algorithm. The main actions that can be distinguished in the event-driven controller algorithm are: *input (lines 1-3)*, *check (lines 4 and 12)*, *calculation (lines 6-9)* and *output (line 11)*. The numbers above coincide with the line numbers of the pseudo-code in section III-D. For the time-driven controller, the check is omitted, but the other actions are identical. The computation times associated with these actions, are indicated by t_{input}, t_{check}, t_{calc} and t_{output}, respectively. The total processing time of the time-driven controller for a 10 seconds experiment (t_{td}^{10}), can be computed as follows:

$$t_{td}^{10} = 10 f_s (t_{input} + t_{calc} + t_{output}) \tag{4}$$

with f_s the sample frequency of the controller.

For the event-driven controller, the total processing time of a 10 seconds experiment (t_{ed}^{10}) is given as:

$$t_{ed}^{10} = 10 f_s (t_{input} + t_{check}) + c^{10}(t_{calc} + t_{output}) \tag{5}$$

with c^{10} the number of control updates over a 10 seconds experiment.

Quantitative estimates of the individual computation times can be obtained from micro measurements, also called benchmark numbers. These measurements give the time duration of individual basic operations, like e.g. the addition of two floating point numbers and depend on the speed of the cpu, memory and on the floating point unit. The tasks that run on the processor can be split up in terms of those basic operations. From this we can obtain the expected computation times of the tasks. For instance, *check* executes two floating point comparisons and one logical operation (OR). From micro-measurements we know that a floating point comparison takes $0.025\mu s$. The time that the logical operation takes can be neglected with respect to a floating point operation. Therefore, t_{check} is estimated to be $0.05\mu s$. As *calc* has to perform a larger number of floating point operations, we estimate t_{calc} to be $0.25\mu s$.

Similarly, t_{input} and t_{output} are estimated at $2.4\mu s$ and $2.1\mu s$ respectively. The relatively large amount of time consumed by an IO-operation is caused by context-switch time and the time it takes to communicate with the slower PCI-bus. In order to perform an IO operation, communication via a device-driver with the FPGA is necessary. A context-switch is made to and from the kernel-space to access the device-driver. t_{input} is estimated somewhat larger than t_{output}, because some additional processing is involved to derive the velocity measurement from the position data.

The times for the various operations can be assumed to be fairly constant. For this specific example, the controller is the only real-time task running and its size allows it to run entirely from cache memory, so no variations are expected.

Using equations (4) and (5), combined with the estimated computation times of the individual tasks, we are able to predict the total computation times of both the time-driven and the event-driven control algorithm for various values of e_T. From the simulation results, depicted in the bottom graph of figure 5, we use the number of control updates as the value for c^{10} in (5). The results are depicted in figure 6 (dashed and solid line).

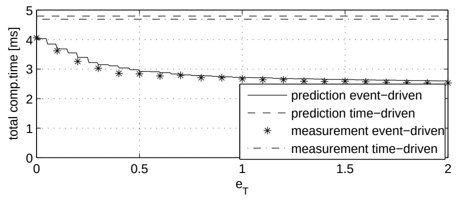

Fig. 6. Experimental results of computation time for time-driven and event-driven controller using various values of e_T for the event-driven controller.

From equations (4) and (5) we can also derive the maximal achievable gain in processing time for the event-driven controller, compared to the time-driven controller. The maximal gain is obtained when we choose $c^{10} = 0$. This implies that no control updates are needed to keep the error within the bounds of e_T. The maximum achievable gain K_{max} in this particular setup is:

$$\begin{aligned} K_{max} &= \frac{10 f_s (t_{input} + t_{calc} + t_{output})}{10 f_s (t_{input} + t_{check}) + c^{10}(t_{calc} + t_{output})} \\ &= \frac{t_{input} + t_{calc} + t_{output}}{t_{input} + t_{check}} \approx 2 \qquad (6) \end{aligned}$$

This can be verified in figure 6, as the time-driven controller uses approximately 5 ms and the event-driven controller 2.5 ms for large values of e_T.

VI. Experiments

A. Processor load measurement

To measure the processor load of the control algorithm, we measure the time the processor needs from the start of *read*, to the end of *write*. For this purpose, we take two time-stamps; the first before line 1 in the pseudo-code, and the second after line 12. By subtracting the first time-stamp from the second, we obtain the elapsed time. From this value, we also need to subtract the time it takes to perform the time measurement itself, as on an x86-compatible cpu, time measurement is not atomic; it takes approximately 1 μs for this particular setup.

In order to measure time, a function is called which reads a time counter and returns its value. Some time will elapse between the *call* and the *read* and between the *read* and the *return*. Figure 7 illustrates this as well as the method to determine the amount of time that is required to measure time. The method consists of two successive 'Gettime' function calls. The time difference of the values returned is the time required to perform one 'Gettime' function. It is assumed that the 'Gettime' functions take an equal amount of time and will not be interrupted. The latter is guaranteed by the real-time operating system by assigning it to the highest priority. The amount of time that is required to measure time is measured each sample moment, next to the two time measurements mentioned above.

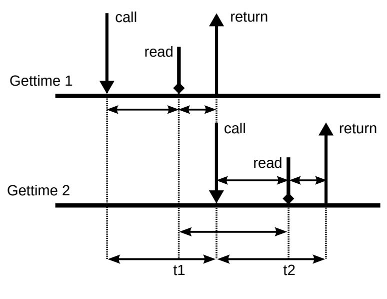

Fig. 7. Illustration of the time measurement.

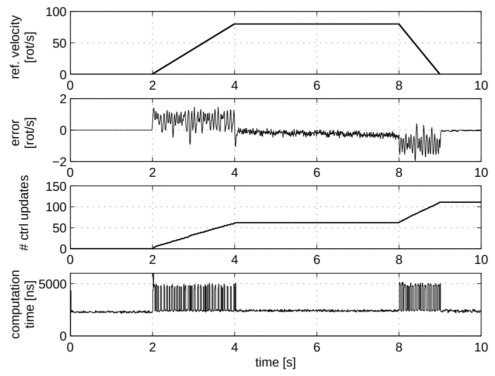

Fig. 8. Experimental results of event-driven controller for $e_T = 0.9$ rot/s.

B. Experimental results

The measurement results of one experiment with $e_T = 0.9$ rot/s are depicted in figure 8. One can compare these plots with the simulation results depicted in figure 4, as the same value for e_T is chosen. The second plot, which depicts the velocity error, shows similar values for the experiment as well as for the simulation. This is also the case for the third plot, in which the number of control updates are plotted. The fourth plot shows the measured execution time of the controller algorithm at each sample time, i.e. every 0.01 second. The offset of 2.4 μs ($t_{input} + t_{check}$) can clearly be distinguished. The reason that the offset differs slightly at periods $[1, 2]$ and $[9, 10]$ is that for those periods, the reference as well as the measured velocity are exactly zero, which involves less computation time for the implementation. The extra time at the moments of the peeks in the plot during non-zero velocity in the reference signal, is the time that is needed for *calc* and *output* ($t_{input} + t_{check} + t_{calc} + t_{output} = 4.8 \mu s$).

The performance results (e_{max}) of 20 experiments for different values of e_T are depicted in the first graph of figure 5, together with the simulation results. The experimental results are very similar to the simulation results and we observe the same lower bound and trend. The maximum error of the time-driven controller experiment was the same as obtained from simulation. The second graph of figure 5 shows the number of control updates for these 20 experiments, together with the simulation results. We observe again very similar results.

The cumulated measured computation time for the same 20 experiments is depicted in figure 6, together with the predicted computation times. This again is depicted for both the time-driven and the event-driven controller. It can be observed that the measurement results are very similar to the predictions.

VII. Discussion

When comparing the results of the time-driven controller with the results of the event-driven controller, we observe a reduction of 30% in the number of control updates, for very low values of e_T ($0 \leq e_T < 0.1$). This resulted in a processing time saving of 13% of the total processing time needed for the control algorithm. For higher values of e_T a reduction could be achieved up to 95% in the number of control updates. This however resulted in a saving of the processing time of only 46%, due to the relative high offset caused by the value of t_{input} (as this has to be performed at 100 Hz in the event-driven controller as well). When we choose for example $e_T = 0.4$ rot/s in the presented application, we already obtain a saving in the processor load of 39%. This only increased the maximum error from 1.2 rot/s to 1.7 rot/s.

Of course, these figures depend heavily on the chosen setup. Important aspects of the setup in this context are: the complexity of the controller algorithm, the processing platform together with communication mechanisms, the reference signal to be tracked and the disturbances acting on the plant. For the experiments we used a very simple (PI) control algorithm, that does not need much processing power to execute (i.e. t_{calc} was small). If we choose a more complex control algorithm, the savings in processing time increase.

If a different processing platform is used on which the input and output actions would need only a very small amount of the processing time, we would have gained up to a factor of 5 in processing time in the presented situation. Indeed, in that case we have that $t_{input} \approx t_{output} \ll t_{calc} \approx 5 \cdot t_{check}$. Therefore, we see that the relation between the processor load and the number of control updates is highly dependent on the chosen processing platform. The presented case is just one typical example out of many possible alternatives. The strength of this paper is that it indicates that the gain in processor load can be predicted for each new situation, in a similar way as presented in this paper. If sensor and actuator data have to be communicated over a network with limited bandwidth, savings of these kinds might be considered as well.

By applying the considered event-driven controller we only decrease the average processor load and not the peak load. Therefore, it should be noted that the processing power that comes available temporarily should also be used to create an advantage for the total system (e.g. the whole paper path). One example is the execution of multiple motor controllers on the same processor. If these controllers are for instance used to control all 5 motor velocities in our paper path setup, we can choose the profiles such that the acceleration and deceleration phases do not overlap. This is a very realistic choice, from a system level power perspective, as it is often required to distribute high power demands over time. When a motor is accelerating or decelerating, it uses more power. Because we know that (for the non-disturbed case) most control updates need to be executed during those phases, the processing power is also nicely distributed among the 5 controller tasks. Another example to reduce the overall processor load, is the case in which soft-realtime tasks (e.g. image processing), running on the same processor, can use the released processing power.

VIII. Conclusions

The contributions of this paper are twofold:

1) The potential of event-driven controllers was validated.
2) The relation between reduced number of control computations and a lower processor load was studied.

We validated for the first time *experimentally* the potential of event-driven controllers. Experiments showed that event-driven controllers can be used in practice to reduce the processing time by a factor of almost 2, when compared with conventional time-driven controllers. This involved only a small degradation of the control performance. We also argued that for the particular controller setup on a different processing platform, even a factor of 5 reduction in processor load could have been obtained, which shows the potential value of event-driven controllers and future research in this domain.

From simulation results, we were able to predict the processor load in the experiment very accurately. This was done with relative low effort, despite of the fact that many complex implementation factors are to be accounted for. For this purpose, micro measurements were used to estimate the processing time of the various tasks of the controller algorithm. The main benefit of the prediction method is that one does not have to actually build the setup to quantify the trade-off in processor load and control performance for the event-driven controllers.

References

[1] Årzén, Karl-Erik (1999). A simple event-based PID controller. In: *Proceedings of the 14th World Congress of IFAC*. Beijing, P.R. China.

[2] Doff, R.C., M.C. Fatten and C.A. Phillips (1962). Adaptive sampling frequency for sampled-data control systems. In: *IRE Transactions on Automatic Control*. vol. AC-7. pp. 38–47.

[3] Franklin, G.F., J.D. Powell and M.L. Workman (1998). Digital Control of Dynamic Systems. *Third edition, MA: Addison-Wesley.*

[4] Henriksson, D. and A. Cervin (2005). Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In: *Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference*, Seville, Spain, December 2005.

[5] Sandee, J.H., W.P.M.H. Heemels and P.P.J. v.d. Bosch (2005). Event-driven control as an opportunity in the multidisciplinary development of embedded controllers. In: *Proceedings of the American Control Conference*, Portland, Oregon, USA, pp. 1776–1781.

[6] Heemels, W.P.M.H. and J.H. Sandee (2006). Practical stability of perturbed event-driven controlled linear systems. *Proceedings of the American Control Conference*, Minneapolis, Minnesote, USA.

[7] 20-Sim, modelling and simulation package (2006). Controllab Products Inc., Enschede, The Netherlands. [online] http://www.20sim.com.