Computer Architectures to Close the Loop in Real-time Optimization

E. C. Kerrigan, G. A. Constantinides, A. Suardi, A. Picciau and B. Khusainov

***Abstract*— Many modern control, automation, signal processing and machine learning applications rely on solving a sequence of optimization problems, which are updated with measurements of a real system that evolves in time. The solutions of each of these optimization problems are then used to make decisions, which may be followed by changing some parameters of the physical system, thereby resulting in a feedback loop between the computing and the physical system. Real-time optimization is not the same as 'fast' optimization, due to the fact that the computation is affected by an uncertain system that evolves in time. The suitability of a design should therefore not be judged from the optimality of a single optimization problem, but based on the evolution of the entire cyber-physical system. The algorithms and hardware used for solving a single optimization problem in the office might therefore be far from ideal when solving a sequence of real-time optimization problems. Instead of there being a single, optimal design, one has to trade-off a number of objectives, including performance, robustness, energy usage, size and cost. We therefore provide here a tutorial introduction to some of the questions and implementation issues that arise in real-time optimization applications. We will concentrate on some of the decisions that have to be made when designing the computing architecture and algorithm and argue that the choice of one informs the other.**

I. What's in a Name?

Suppose you need to drive from your home to an important meeting. You would not be happy if your in-car navigation system told you to take turns after you have passed the points where you should have turned, causing you to be lost and late. As you have probably also discovered, the traffic situation on even a short journey can change very fast — the fastest trip at the beginning of the journey may have resulted in you being stuck in a traffic jam and arriving much later than expected. If only you had upgraded your navigation system with the real-time traffic option, which would have continuously monitored the situation and updated your journey before you got stuck on the road with everybody else...

The key challenge in real-time optimization applications is clearly how best to deal with *time*. In the real world, correctness of a computation is a function of time. The late arrival of a computation might be undesirable or, in some cases, even fatal. The opposite is also true. If a result arrives too soon, then it might be invalid when it is needed if the situation has changed. The computation would therefore have to be restarted, meaning that the previous processing, communication and storage needed to produce the result would have been unnecessary.

The authors are with the Department of Electrical & Electronic Engineering, Imperial College London, SW7 2AZ London, UK. E. C. Kerrigan is also with the Department of Aeronautics. e.kerrigan@imperial.ac.uk

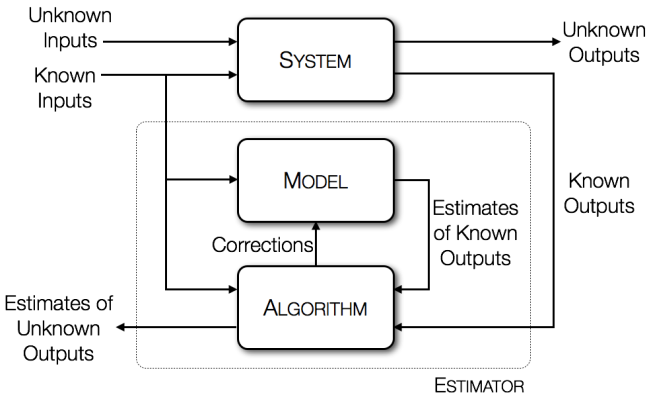

Fig. 1. Block diagram of a model-based estimator.

The reason why time is an issue, is because of *uncertainty*. Without uncertainty, one could set up and solve a single optimization problem, whose solution would be valid for all time and there would be no need to take new measurements of the system. In real-time optimization, on the other hand, measurements of an uncertain system or process are used to formulate and solve a sequence of optimization problems. By using the latest measurements, one can ensure that the solution to the optimization problem remains valid. Without uncertainty, measurements would not be necessary. This is why real-time optimization is more than just 'fast' or 'robust' optimization — a computation that arrived too soon may not be valid in the future and a solution that does not require measurements might result in an unnecessarily over-engineered system.

Broadly speaking, real-time optimization applications can be divided into two categories, namely signal processing and control. In the former, the goal is usually to estimate (or learn) some variables of a system or process. In the latter, the aim is to manipulate some inputs of a system to ensure that certain variables satisfy some constraints or that some cost function is minimized. In both cases, feedback is the key tool used to manage the effect of uncertainty.

Consider Figure 1, which is a block diagram for a typical model-based signal processing (or machine learning) algorithm. Measurements of an actual system are compared against the predictions of an internal model. If there is a difference, then some internal model variables are updated until the predictions and measurements (and/or their differences) satisfy given constraints. Note that there is feedback (a closed loop) between the model and the algorithm, i.e. the algorithm is

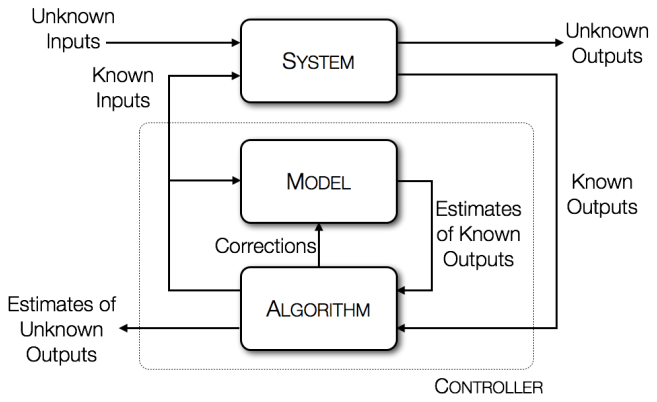

Fig. 2. Block diagram of a model-based controller.

trying to control the model to ensure that the model and system outputs satisfy some criteria.

One popular type of optimization-based signal processing method, of which the popular Kalman filter is a special case, is Moving Horizon Estimation (MHE) [1]–[3]. In MHE, only a finite amount of measured data is kept in memory. At each sample instant, the oldest measurements are used to update the data of an optimization problem before being removed in order to make space for new measurements. Once the data is updated, the optimization problem is solved in order to produce estimates of the physical system state sequence, parameters, unmeasured inputs and measurement noise.

It is interesting to compare Figure 1 with Figure 2, which is a block diagram of a typical control (or automation) system. Measurements of a physical system or process are compared against the predictions of an internal model. If there is a difference, then some model variables and the inputs to the actual system are updated until the predictions and the measurements satisfy some constraints. There is feedback between the algorithm and the system as well as the algorithm and the model. Note also that there is a difference between Figure 1 and Figure 2 — in Figure 1 the known input of the system is also an input of the algorithm, whereas in Figure 2 the known input of the system is an output of the algorithm.

Model Predictive Control (MPC) is a very popular optimization-based control method [2], [4]. In MPC, only part of the solution to an optimization problem is used to update the control input after each sample instant. At each sample instant, a new measurement is taken in order to update the data to the optimization problem, often after first solving an estimation problem. The optimization problem is then solved with the new data in order to compute a new control update.

There has been considerable research in the MPC and MHE communities in the last few decades in the development of efficient algorithms for real-time optimization [4], [5]. Much of this literature abstracts away the hardware details of the computing system on which the algorithm will be implemented. However, recently there has been a growing interest in understanding how best to exploit the particular features of certain processors for real-time optimization, which has opened up new application areas for real-time optimization.

The aim of this paper is therefore to give a tutorial introduction on computer architectures and in order to help inform the choice real-time optimization algorithm, and vice versa. The development has been kept at a high level in order to allow for a relatively large readership, which will hopefully include advanced undergraduates, practising engineers and expert researchers who would like to know a little bit more about the state of the art and future trends.

Much of the discussion here also applies to numerical algorithms in general and not just real-time optimization. In many cases it makes sense to first explore the limits of existing methods and architectures in order to justify the need for developing new, tailor-made solutions. When doing fundamental research on any specific topic, such as real-time optimization, it is also often the case that new results and design methods of more general interest are developed.

The paper does not attempt at providing mathematically rigorous answers or design procedures to well-defined questions on how best to match current algorithms to current hardware, or vice versa. This is partly because the combination of algorithms, hardware and questions are too numerous for an introduction to this topic, but mostly because this is a significant research activity still in its early stages. Instead, we will give a flavour of the variety of design choices and trade-offs that could be made and have included references to selected papers that aim to provide answers to particular questions.

Section II argues that, when assessing the performance of a real-time optimization algorithm, one should not consider the behavior of the algorithm in isolation, but instead consider the performance of the combined cyber-physical system. Section III defines a general class of mathematical problems that arise in real-time optimization applications, where the sequence of optimization problems is an explicit and/or implicit function of time and the outputs of the resulting cyber-physical system. A brief introduction to the main concepts in computer architecture, relevant to real-time optimization, are given in Section IV. Section V discusses some of the advantages and disadvantages of current hardware and software technologies for implementing real-time optimization algorithms. A very brief look into the future of computer architectures is given in Section VI before presenting some concluding remarks in Section VII.

II. Time and Uncertainty in Cyber-Physical Systems

Suppose that, as depicted n Figure 3, we have a causal, dynamic system

$$P : (u, w) \mapsto (y, z)$$

that evolves in (real-world) time with measured output $y : \mathbb{R} \to \mathcal{Y}$ and where $u : \mathbb{R} \to \mathcal{U}$ are (control) inputs that can be changed/manipulated in time. The output $z : \mathbb{R} \to \mathcal{Z}$ includes measured and unmeasured variables that we wish to

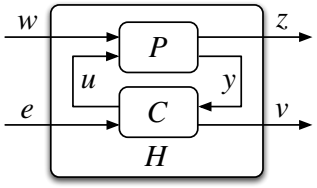

Fig. 3. Block diagram of a cyber-physical system

estimate, optimize, constrain or regulate. The input $w : \mathbb{R} \to \mathcal{W}$ represents unknowns, which include measurement noise, unmeasured disturbances, time-varying set-points that are not known in advance, uncertain parameters and unmodelled dynamics.

The computing system, which contains an implementation of the real-time optimization algorithm, is a strictly causal dynamic system

$$C : (y, e) \mapsto (u, v),$$

where the input $e : \mathbb{R} \to \mathcal{E}$ represents computational errors, e.g. due to finite precision arithmetic errors, early termination of the algorithm or because the solver could only compute a locally optimal point. The output $v : \mathbb{R} \to \mathcal{V}$ contains results of computations, including estimates of the accuracy and precision of these computations.

One of the main points to note is that the performance and robustness of a real-time optimization algorithm should be evaluated by including the evolution of the physical system in the analysis. In other words, the correctness of the real-time optimization algorithm should be based on whether the cyber-physical system

$$H : (w, e) \mapsto (z, v)$$

satisfies given constraints on performance and robustness.

The abstract ideas above can be illustrated with a simple optimal control example. Consider the problem of computing the input trajectory (with $\mathcal{U} := \mathbb{R}$) to an integrator such that the output (with $\mathcal{Y} := \mathbb{R}$) is driven from zero to one in one second, while minimizing the integral of the square of the input. Suppose a given computer system (i.e. algorithm and hardware) is guaranteed to produce a feasible trajectory to this problem in δ seconds and that the input is arbitrarily set to zero until a result is available. The optimal control problem can therefore be defined as solving the following infinite-dimensional optimization problem:

$$(u^*(\cdot, \delta), y^*(\cdot, \delta)) := \arg\min_{(u,y)} V(u) \tag{1a}$$

subject to

$$y(0) = 0,\ y(1) = 1, \tag{1b}$$

$$u(t) = 0,\ \forall t \in [0, \delta), \tag{1c}$$

$$\dot{y}(t) = u(t),\ \forall t \in [0, 1) \text{ a.e.}, \tag{1d}$$

where the cost function

$$V(u) := \int_0^1 u(t)^2 dt. \tag{1e}$$

It is possible to prove that the optimal input trajectory is given by

$$u^*(t, \delta) = \begin{cases} 0 & \forall t \in [0, \delta) \\ 1/(1-\delta) & \forall t \in [\delta, 1) \end{cases} \tag{2}$$

and that the minimum of the cost function is given by

$$V^*(\delta) := V(u^*(\cdot, \delta)) = 1/(1-\delta). \tag{3}$$

Note that V^* is a continuous, monotonically increasing function of the delay δ.

Of course, this is a trivial example for which it is arguably not necessary to use a numerical method and a computer to produce the solution. However, it will suffice for our purpose in making an important point about real-time optimization, namely that the correctness of a computation should be expressed as a function of time.

Consider, for the sake of illustration, that the computer generates a sequence of feasible, but sub-optimal input trajectories of the form

$$\tilde{u}(t, \delta, \mu) := \begin{cases} 0 & \forall t \in [0, \delta) \\ \mu & \forall t \in [\delta, 2\delta) \\ (1-\mu\delta)/(1-2\delta) & \forall t \in [2\delta, 1) \end{cases} \tag{4}$$

where $\mu \in \mathbb{R}$ changes at each iteration of the algorithm. The resulting value of the cost function for this trajectory is

$$V(\tilde{u}(\cdot, \delta, \mu)) = \mu^2\delta + \frac{(\mu\delta - 1)^2}{1 - 2\delta}. \tag{5}$$

Note that if $\mu = 1/(1-\delta)$, then the trajectory $\tilde{u}$ is optimal, i.e. $V(\tilde{u}(\cdot, \delta, 1/(1-\delta))) = V^*(\delta)$.

Suppose that it takes $\delta = 0.4\,\text{s}$ for the computer to find the optimal input sequence, i.e. $\mu \approx 1.67$ after termination. The optimal value of the cost function in this case is $V(\tilde{u}(\cdot, \delta, \mu)) = V^*(\delta) \approx 1.67$. It follows from (5) that a smaller latency δ results in a lower value for the minimum. Hence, it is possible that a sub-optimal trajectory to the optimal control problem (1) with a smaller δ will result in a lower value of the cost function than an optimal trajectory with a larger δ.

Consider therefore the scenario where it is known in advance that the algorithm will be terminated prematurely at $\delta = 0.2\,\text{s}$ and that the data of the optimal control problem (1) is updated to reflect this. Suppose it turns out that the resulting sub-optimal trajectory[1] has $\mu = 0.5$, hence the resulting value of the cost function is $V(\tilde{u}(\cdot, \delta, \mu)) = 1.4$. In other words, a trajectory that is sub-optimal for $\delta = 0.2\,\text{s}$ results in a lower cost than a trajectory that is optimal for $\delta = 0.4\,\text{s}$.

This example demonstrates that, because the physical system continues to evolve with time while the computation is being carried out, it might be better to implement a result with some numerical error (such as sub-optimality) instead of an exact or more accurate answer at a later time. Furthermore, the physical system is constantly subject to unknowns and there is nearly always some modelling error. One might therefore be

[1]The optimal trajectory for $\delta = 0.2$ has $\mu = 1.25$ with $V(\tilde{u}(\cdot, \delta, \mu)) = V^*(\delta) = 1.25$.

able to use a cheaper microprocessor and simpler algorithm, but increase the measurement, actuation and communication rate, while improving the performance as judged from the perspective of the continuous-time physical system.

Methods for real-time optimization build on 'efficient', 'fast' or 'robust' optimization theory. However, because of the presence of uncertainty and the fact that time does not stop, real-time optimization has its own unique set of features, problems and solutions, which do not necessarily arise in other areas of optimization.

III. Real-time Optimization

In real-time optimization, one has to solve a sequence of optimization problems where only part of the problem data changes explicitly or implicitly with time. Each problem in the sequence is typically of the form

$$\min_x \{J(x,d,t) \mid x \in X(d,t),\ d = D(y,v,t)\}, \qquad (6)$$

where $t \in \mathcal{T} \subset \mathbb{R}$ denotes (real-world) time and $x \in \mathcal{X}$ is the decision variable, which can include estimates of the parameters of the physical system, as in signal processing applications, and/or variables that are to be fed back into the physical system, as in control applications. The time-varying component of the problem data d is given by a function $D : \mathcal{Y} \times \mathcal{V} \times \mathcal{T} \to \mathcal{D}$ of the measured outputs, previous computations and time, while $J : \mathcal{X} \times \mathcal{D} \times \mathcal{T} \to \mathcal{J}$ is the cost function and $X : \mathcal{D} \times \mathcal{T} \to \mathcal{X}$ defines the set of feasible points.

An important point is that the data d often only changes slightly from one time instant to the next. Furthermore, in many practical applications, the size of this difference tends to zero as the difference between time instants tend to zero. This fact can be exploited at both the algorithmic and hardware level to design efficient optimization solvers. For example, one could modify previous computations to 'warm-start' the optimization solver with a good initial point in order to reduce the number of iterations, minimize the amount of access to slower, off-chip memory and decrease the bandwidth required for communication between various sub-systems.

A. Dynamic Optimization

The optimization problem (6) may be defined in a variety of ways. In many cases, such as MPC and MHE, a dynamical model of the physical system is employed to define an optimal control or estimation problem, which can be mapped into an equivalent infinite- or semi-infinite dimensional optimization problem. This problem is usually discretized in order to formulate a finite dimensional optimization problem, which can then be solved using a numerical method. This so-called *dynamic optimization* problem often has a particular structure, which can be exploited when designing and implementing a real-time optimization solver [5], [6].

In many cases, the dynamic optimization problem can be written in the form:

$$\min_{q \in \mathcal{Q}, s \in \mathcal{S}} \ \sum_{i=0}^{N-1} \ell(q_i, s_i, s_{i+1}, i, d, t) \qquad (7a)$$

subject to

$$f(q_i, s_i, s_{i+1}, i, d, t) = 0,\ i = 0, \ldots, N-1, \qquad (7b)$$

$$g(q_i, s_i, s_{i+1}, i, d, t) \leq 0,\ i = 0, \ldots, N-1, \qquad (7c)$$

where the problem data d changes at each time instant t and ℓ is the stage cost. Estimates of the state or parameters of the dynamical system are usually included in the sequence $s := (s_0, \ldots, s_N)$, while the sequence $q := (q_0, \ldots, q_{N-1})$ often includes estimates of the unmeasured and manipulated/control inputs. The equality constraints defined by the function f often arise due to the dynamic model of the physical system and path constraints, while the inequality constraints defined by g capture physical, performance, safety and other constraints.

A key observation here is that (7) can be re-written as an optimization problem for which the cost and constraint functions are block separable, provided the decision variable is defined by interleaving the components of s and q as $x := (s_0, q_0, s_1, q_1, \ldots, s_{N-1}, q_{N-1}, s_N)$. If ℓ, f and g are sufficiently differentiable, then the Jacobian and Hessian of the optimization problem are block tridiagonal. This structure can be exploited with efficient implementations of direct [7] or iterative [8] sparse linear algebra algorithms [5], [6], [9].

B. Matching Algorithms and Hardware

The sequence of problems that need to be solved at each time instant can take many different forms, from unconstrained linear least squares problems to constrained nonlinear optimization problems, such as quadratic or semi-definite programs. Since the performance of an algorithm is a function of the computer architecture of the system on which it is implemented, the choice of one has to inform the other and vice versa. Examples of this include:

- Direct methods for solving linear systems, such as Cholesky factorization, often include many square root computations, which are significantly more complicated to realise in hardware than addition, and can be very difficult to parallelise. Some iterative methods, such as the conjugate gradient method, have relatively few square root computations and are also much easier to parallelise. On the other hand, Cholesky factorization is often less sensitive to truncation and round-off errors than the conjugate gradient method.
- The matrices that are generated at each iteration of an interior point method often have the same size, whereas this is not the case with active set methods [10]. It can be easier to map algorithms to hardware if the matrices are of the same size. On the other hand, in some situations it might be better to generate a sequence of matrices whose dimensions are time-varying, but smaller.
- Newton-based methods involve computing second-order derivates and solving a sequence of linear equations. Though second-order methods have superior convergence rates compared to first-order methods, which only use gradient information, first-order methods are often easier to implement and might be preferable to second-order methods in certain applications [11].

Algorithmic parameters could also affect the required computational resources and hence the overall performance of the cyber-physical system. For example,

- Including a high-order model of the dynamics of the system in the optimization problem might provide more accurate answers, but is likely to require more computational resources. Since the actual system is subject to disturbances inbetween sample instances, it might be better to use a low-order, less accurate model if the time between samples can be reduced, so that new measurement can be used to correct for errors.
- One might be able to trade off the number of iterations and accuracy of computations at various levels in an algorithm against the final result. For example, with inexact Newton methods the resulting linear equations are only solved approximately using iterative solvers, such as the conjugate gradient method. This might result in more Newton steps, but since each iteration is more efficient than for an exact Newton method with Cholesky factorizations, the overall time taken by an inexact method might be less than that for an exact method [12].

IV. Computer Architectures

The computing system is a standalone computer or embedded system that executes the optimization algorithm and communicates with the physical or other computing systems. This is a causal system where the following tasks are executed in a certain amount of time:

- *acquisition phase*: read the output data from the physical system or neighboring computing system;
- *processing phase*: execute the algorithm, processing input data together with the data stored locally;
- *output phase*: output the results of the computation to the physical system and send information to the neighboring computing systems.

Figure 4 shows the resulting computer architecture, which consists of three main sub-systems: processor, communication and storage. The processor (Section IV-A) is the computational unit of the system, where the optimization algorithm runs. The communication module (Section IV-B) transfers data in and out via sensors, actuators and network interfaces. Finally, the storage module (Section IV-C) is an external memory that allows one to store algorithm setup data, intermediate data and results whenever this data does not fit into the limited-size internal processor memory.

When designing the computer architecture, there is a large choice of processor, communication and storage modules available to purchase and connect. However, the computer architecture is tightly coupled with the optimization algorithm when the objective is to build a high performance system. In Section IV-D we will discuss this relationship and the main design trade-offs involved. We will also introduce a systematic co-design methodology to address the problem.

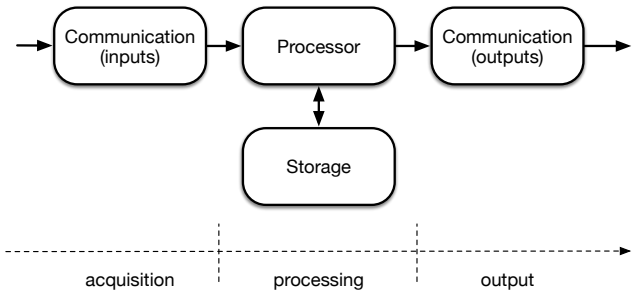

Fig. 4. Computing sub-systems

A. Processing

The processor is the computational device that executes the optimisation algorithm. The performance of the algorithm, in terms of execution delay and quality of the results, depends on how well the algorithm can be mapped onto the available processor's computing resources. Many design techniques can be embraced to achieve a target computing performance. They range from inferring parallellization and pipelining data instruction processing (Section IV-A.1) to selecting the appropriate numerical precision and accuracy (Section IV-A.2).

1) Parallel Processing: The execution time of an optimization algorithm depends on how the algorithm itself can be mapped onto the processor architecture. A classical approach to reduce the execution time is to perform many computations simultaneously, i.e. in parallel. However, this technique can only be embraced if the target processor architecture supports this and the algorithm allows concurrent computations. For example, the multiplications in a dot product can be parallelized, but not conditional branches. On the other hand, a multicore CPU or a GPU have a specific hardware architecture that enables these types of parallellizable operations, but many micro-controllers do not.

Parallelism can be exploited at different levels within an algorithm:

- *Task level* is the case when the calculations can be split across multiple processing units and processed at the same time using either the same or different sets of data. Two processor architectures are suitable for this type of parallelism: i) multicore CPU-like multiple instruction, multiple data (MIMD), where each computational unit can perform different operations on different data sets; ii) GPU-like single instruction, multiple data (SIMD), where the same operation is executed at the same time over many data sets. As an example, the former is suitable to implement linearization algorithms and the latter to implement dense matrix-by-vector multiplication.
- *Instruction or data pipelining* is a technique where a set of instructions/operations are arranged into a sequence of dependent steps executed concurrently and by different hardware circuits. This approach increases the throughput, but does not reduce the execution time. Pipelining at instruction level is available in most CPU-

Algorithm 1 Example algorithm

$z = x + y$
$w = \alpha \cdot x$
$v = w \odot x$

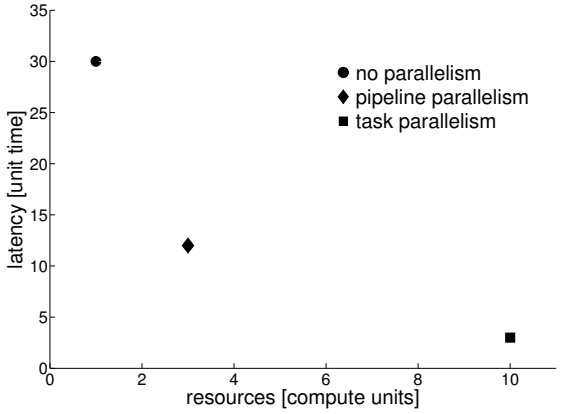

Fig. 5. Comparison of different types of parallelism on the resources usage (compute units) and execution time (latency) of Algorithm 1 when $n = 10$.

and GPU-like architectures and pipelining at data level is available in FPGAs.

- *Bit level* exploits parallelism for the execution of a single operation (i.e. addition, multiplication, etc.) so that the computation is completed in one clock cycle. Nowadays, this type of parallelism is available on most processor architectures.

Consider Algorithm 1, composed of a sequence of three vector operations (e.g. operations within the loop of a gradient-based optimization algorithm) in order to illustrate the advantages/disadvantages of task and pipeline parallelism: where α is a scalar and $x, y, z, v \in \mathbb{R}^n$. Figure 5 shows resources usage (compute units) and algorithm execution time (latency) for an implementation without parallelism and with task and pipeline level parallelism. It could be noted that:

- The case with no parallelism is the slowest, but uses just one compute unit. Each element of the resulting vector will be computed every 3 time steps and all the results will be computed in $3n$ time steps. This would be the case when a single-core CPU or a micro-controller is used.
- Task level parallelism is the fastest. In the best case, this takes a number of steps equal to the number of operations (3 for our example) to provide the result, but requires as many compute units as the dimension n of the the vectors. A multi-core CPU or a GPU processor architecture supports this type of parallelism.
- Pipeline parallelism is a compromise approach. This provides a smaller delay and higher throughput (1 unit time), compared to the sequential case, while at the same time requiring fewer resources compared to task parallelism. This approach is commonly used in FPGA processors.

2) Number Representation: Choosing an appropriate way for representing numerical data may have a significant effect on the speed, robustness and cost of the entire cyber-physical system. Conventional optimization platform design approaches propose developing the high-level algorithm, followed by hardware implementation with a certain type of number representation. This decoupled approach leads to sub-optimal results, since some of the hardware platform capabilities are left unexploited [13].

The vast majority of today's computational platforms use either *floating* or *fixed point* arithmetic. A binary representation of a floating point number consists of three components: sign bit, mantissa and exponent, which defines the location of the radix point relative to the mantissa and hence determines the dynamic range. The term *dynamic range* is usually defined as the ratio of the maximal to minimal representable numbers. The first bit of the mantissa is assumed to be 1 and not explicitly stored in the *normalized* form. In contrast to floating point, fixed point data representation allocates a fixed number of bits for the integer and fraction parts without any flexibility in terms of the radix point location.

The high dynamic range of floating point arithmetic is achieved at the cost of increased latency, silicon usage and power consumption. Since exponents of two numbers are not the same, in general, even simple addition is preceded by denormalization and followed by normalization. For example, in some FPGAs fixed point addition of 32 bit numbers requires one clock cycle and 32 Lookup Tables (LUTs), whereas addition of IEEE 754 single precision floating point numbers with the same bit length would employ 500 LUTs and take 6 clock cycles at 350MHz clock rate [14]. On the other hand, fixed-point arithmetic introduces additional overflow and round off errors. Overflow errors happen due to lack of dynamic range and can be avoided by pre-calculating the largest possible absolute value and allocating an appropriate scaling factor. It is also essential to prove stability of iterative optimization algorithms under the presence of round-off errors. Even transforming the original problem to fixed point representation may lead to loss of favorable properties of the objective function, such as convexity, or distort the set of feasible points [15].

Depending on the application, it might be justifiable to use other types of data representation, other than fixed and floating point. For instance, a logarithmic data representation is well suited for multiplication and division operations, while preserving high dynamic range [16], [17]. Another example of alternative arithmetic is dual-fixed point, which combines the advantages of fixed and floating point computations [18].

Figure 6 demonstrates how the number of fraction bits for a fixed point data representation affects algorithm execution time, resource utilization and closed-loop performance of the entire cyber-physical system. The results were obtained from hardware-in-the-loop (HIL) simulations of a fast gradient-based [15] model predictive controller for a mass-spring-damper system with $n = 10$ states and $m = 5$ inputs. The

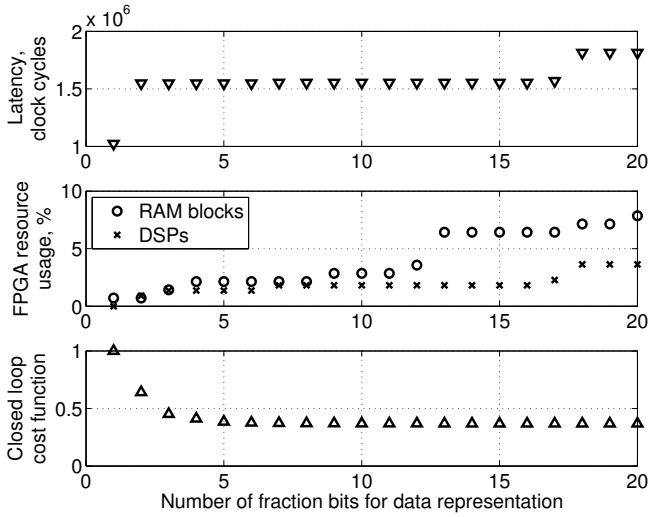

Fig. 6. Latency of a fast gradient optimization-based controller, FPGA resource usage and closed-loop system performance as a function of number of fraction bits. The number of integer bits is fixed to 8. A lower closed-loop cost implies better performance.

plant dynamics

$$x_{k+1} = Ax_k + Bu_k \tag{8}$$

were simulated on a desktop computer while, the controller was implemented on the programmable logic of a Xilinx Zynq-7020 system-on-a-chip. The optimal control problem for the horizon N and the initial condition $\hat{x}$ was formulated as follows:

$$\min_{\substack{u_0 \dots u_{N-1} \\ x_0 \dots x_N}} \frac{1}{2} x_N^T P x_N + \sum_{k=0}^{N-1} (\frac{1}{2} x_k^T Q_d x_k + \frac{1}{2} u_k^T R_d u_k) \tag{9a}$$

subject to

$$x_0 = \hat{x}, \tag{9b}$$

$$x_{k+1} = Ax_k + Bu_k, \quad k = 0, 1, \dots, N-1 \tag{9c}$$

$$u_{min} \leq u_k \leq u_{max}, \quad k = 0, 1, \dots, N-1 \tag{9d}$$

where $Q_d \in \mathbb{S}_+^n$, $R_d \in \mathbb{S}_{++}^m$ and $P \in \mathbb{S}_{++}^n$ are state, input and terminal penalty matrices accordingly. $\mathbb{S}_{++}^n (\mathbb{S}_+^n)$ denotes a set of positive (semi-)definite matrices of size n. Performance of the cyber-physical system was measured with the closed-loop cost function:

$$V(\mathbf{u},\mathbf{x}) = \sum_{k=0}^{N_{sim}-1} (\frac{1}{2} \mathrm{x}_k^T Q_d \mathrm{x}_k + \frac{1}{2} \mathrm{u}_k^T R_d \mathrm{u}_k) \tag{10}$$

where the input sequence $\mathbf{u} = [\mathrm{u}_1^T, \mathrm{u}_2^T, \dots, \mathrm{u}_{N_{sim}-1}^T]^T$ and state sequence $\mathbf{x} = [\mathrm{x}_1^T, \mathrm{x}_2^T, \dots, \mathrm{x}_{N_{sim}-1}^T]^T$ were obtained from HIL simulations.

The notable point on the graph is that after reaching a certain number of bits (6 bits in this case), no significant performance improvement is detected. Hence, there is no need to waist resources by introducing redundant precision. This example illustrates that number representation for real-time solvers should be chosen with respect to the closed-loop system, rather than just considering the accuracy of the optimization solver on its own.

B. Communication

The communication sub-system is responsible for communicating between:

- the computing system and the *physical world* to sample the current system states and provide the control action in control applications. Sampling the physical system by taking measurements affects the optimization algorithm under many aspects. First of all, the sampling action involves reading the *analog* output of the physical system, which evolves in time. Therefore, this sampling should ideally be as fast as possible in order to acquire the exact value. Secondly, the sampled analog value is quantized and converted into digital form. This process introduces an error in the measurement that must be considered during the design of the optimization algorithm and is also one of the reasons why feedback is used.
- computing *sub-systems*. This is a communication channel that enables the data movement between sub-systems, such as storage and processors or between processors for a multi-processor system. Because the main role of this link is to transfer data in and out of sub-systems, it is fundamental as to how much data can be moved in a unit of time (*data transfer rate*) and how much time data takes to move from one sub-system to another (*latency*). The ideal case would be to have the latency the small as possible and the transfer rate as high as possible.
- *external systems*, such as other nodes in a distributed optimization network. This communication link shares the same features of the one between sub-systems, but with the difference that the data has to be sent to a remote location. Thus, more energy is required when moving data.

C. Storage

The storage sub-system is an external memory used to store data needed during the optimization algorithm processing when the processor internal memory is not enough. Many storage types exist, and the selection of the right one is crucial for a real-time computing system. The main parameters that describe a storage system are:

- *Size*: how much data can be stored.
- *Latency:* time taken to read/write a data from/to storage.
- *Data transfer rate*: amount of data transferred in a unit of time in/out of storage.

These performance parameters vary according to the storage type. As an example, Table I reports the typical current performance for three classes of storage systems: DDR3 RAM memory, solid state drives (SSD) and magnetic hard disk drives (HDD).

Firstly, it should be noted that there is a trade-off between storage size and speed (both latency and data rate): a big

TABLE I
STORAGE SYSTEM PERFORMANCE

Storage type	Size	Latency	Data transfer rate
DDR3 RAM memory	10 GB	1 ns	16 GBps
SSD	500 GB	100 μs	250 MBps
Magnetic HDD	1 TB	3 ms	60 MBps

storage system is much slower than a small one. On the other hand, it is also important to keep the processor fed with data to process and not to stall the computation because of missing data. Data access speed depends on the algorithm and how the data is stored in memory. As an example, consider the elements of a vector stored in contiguous memory areas. If the algorithm accesses the vector elements in the same sequence as they have been stored, it makes sense to have a high data transfer rate. This will be the case when dense linear algebra is involved. On the other hand, if the algorithm accesses the vector elements in a random order, as is often the case in sparse linear algebra, the data transfer rate will be lower and it might happen that the processor is idle for a significant amount of time.

D. Design Trade-offs

Since the optimization algorithm and computational hardware are tightly coupled with each other, sequential design of the algorithm and hardware may lead to suboptimal designs. This is illustrated by the following example.

Consider a cost function

$$V(x_s, x_h) := V_s(x_s) + V_h(x_h) + V_{coupling}(x_s, x_h) \quad (11)$$

that reflects the performance of the closed-loop system as a function of software x_s and hardware x_h design variables (lower cost implies better performance). V_s and V_h are software and hardware terms, respectively, whereas $V_{coupling}$ defines the interaction between the algorithm and computational platform. Conventional sequential design will propose minimizing V_s over x_s to design an algorithm, followed by minimizing V_h over x_h to implement the algorithm on hardware. Since the interaction term is not taken into account, the performance of the optimization solver cannot be expected to be as good as possible. Minimizing $(x_s, x_h) \mapsto V(x_s, x_h)$ simultaneously with respect to x_s and x_h is the only way to achieve optimal performance. However, in practice it is extremely difficult both to formally define (11) and solve the corresponding optimization problem.

Moreover, optimizing the performance of a closed-loop system is usually accompanied with optimizing computational resources, namely time, energy and space. For that reason, defining the co-design problem as a multi-objective optimization (MOO) problem appears to be a natural way of formalizing the design trade-offs [19]:

$$\begin{aligned} \min_{x_s, x_h} \quad & \begin{pmatrix} V(x_s, x_h) \\ T(x_s, x_h) \\ E(x_s, x_h) \\ S(x_s, x_h) \end{pmatrix} \\ \text{subject to} \quad & x_s \in X_s \\ & x_h \in X_h \end{aligned} \quad (12)$$

with T, E and S defining computational time, energy and space, respectively. Admissible sets X_s and X_h for software and hardware design variables reflect the design constraints.

The MOO problem (12) can be tackled in at least two ways. The first approach is integrated generating and choosing (IGC). The method proposes minimizing an aggregate objective function (AOF) with respect to the constraints in order to obtain a Pareto optimal solution (i.e. a point where it is impossible to improve one objective function without worsening another). The most straightforward choice of AOF is a weighted sum of all objectives. However, in practice AOF cannot guarantee completeness nor evenness of the Pareto set, which motivates the development of two-phase methods, such as generate first-choose later (GFCL). GFCL implies first generating a set of Pareto optimal points so that the designer can make the final decision by choosing one single point from the set [20], [21].

Regardless of the method that is used for generating the Pareto sets, the MOO problem is usually reformulated as a sequence of single objective problems. The decision variables (x_s and x_h) for those problems can be both continuous (discretization frequency, termination tolerance, chip clock frequency) and discrete (parallelization level, pipeline depth, model order), which makes solving the optimization problem non-trivial. Furthermore, derivative information is not always available for all objectives functions of (12) or this information might be unreliable. Evaluation of an objective function itself can be a time consuming task in some cases. For instance, depending on the complexity of the circuit it may take up to several hours to perform a circuit synthesis and hence evaluate the time, energy and space taken by the hardware platform. Another issue is uncertainty, e.g. the same high level code might be compiled in different ways depending on many factors, including the vendor's software version. As a result, it turns out that computing derivatives can be expensive, unreliable or even impossible. Taking into account all these challengning properties, the most promising robust option to solve MOO design problem (12) is to use derivative-free optimization [22].

It is theoretically possible to build an analytical model describing the relationship between design variables and objectives. Once the model is obtained the MOO can be tackled by suitable algorithms [22]. However, in practice it is extremely difficult to analytically describe the design objectives. To overcome this problem, dedicated expensive MOO algorithms can be employed. Expensive MOO algorithms accept the fact that it is extremely costly to evaluate the objective functions and efficiently utilize limited amount

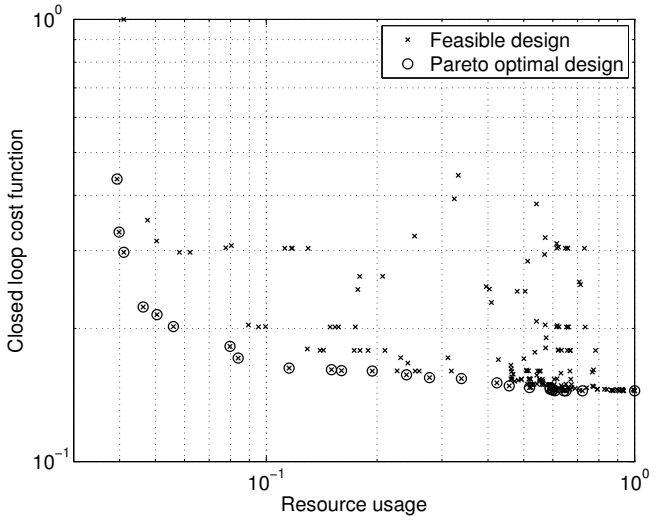

Fig. 7. Trade-off between computational hardware resource usage and closed-loop performance of a cyber-physical system. Both functions are normalized to 1 with respect to maximum values. Resource usage is defined as arithmetic mean of flip-flops, lookup tables, BRAMs and DSPs relative utilization.

of evaluations to build a model and suggest the point for consecutive evaluation [23].

Consider an example of a co-design problem in relation to a model predictive controller (Figure 7). The testcase is similar to the one from Section IV-A.2. The controller horizon length and number of fraction bits for data representation were chosen to be software and hardware design variables, respectively. Two objectives that have to be traded-off against each other are the closed-loop cost function and FPGA resource usage. The plot demonstrates all possible designs, highlighting Pareto optimal solutions. Note that the Pareto frontier is not necessarily convex.

V. Current Technologies

We give a brief overview of the advantages and disadvantages of current processors and software tools for implementing real-time optimization based algorithms.

A. Processors

We discuss five types of processor: microcontroller, multi-core central processing units (CPU), graphics processing units (GPU), field-programmable gate arrays (FPGA) and programmable logic controllers (PLC). These processor types are compared in Table II.

1) Microcontroller: A microcontroller is a single chip that contains a memory unit, an input-output system and a processing unit. The aim of a microcontroller architecture is to provide low latency access to the input-output system at the expense of the capacity of the memory and the computational power of the processing unit.

For example, the microcontrollers of the Atmega family [24] contain a special analog-to-digital signal converter for measuring temperature and four interfaces to send data to other computing devices. However, they can only execute 20 million instructions per second, which is about 150 times fewer instructions per second than a smartphone CPU [25]. The device with the largest memory in this family of microcontrollers has a total capacity of 2 KB, just enough to store a square 27x27 matrix of real numbers. Nonetheless, these microcontrollers are at the core of the popular Arduino boards.

Processors with such small memories can be used for real-time optimization in practice. A microcontroller is suitable for the task if the optimization problem fits into the memory system and if the required computational latency is more than a millisecond. For example, a primal-dual interior point method for the control of an artificial pancreas on a microcontroller is reported in [26].

The power consumption of a microcontroller can be in the order of a mW [24]. This is about two hundred times smaller than that of a smartphone CPU [25]. Recent microcontroller architectures [27] have more computationally powerful processing units than in the past. These architectures are capable of doing floating-point arithmetic operations, so their computational performance is measured in floating-point operations per second (FLOPS). The best computational performance of microcontrollers is in the range of the hundreds of millions of FLOPS. Also, recent microcontroller architectures can carry out more computational tasks at the same time. However, more computationally powerful processing units come at the price of a power consumption that is closer to that of a smartphone CPU.

2) Multi-core Central Processing Unit (CPU): A multi-core CPU is a computing device composed of two or more processing units. Multi-core CPUs can be found in desktop and laptop computers, but also tablet computers and smartphones. The design aim of a multi-core CPU is to execute many computational tasks at the same time, putting into effect a type of execution called multiple-instruction-multiple-data [28]. For example, a multi-core CPU can perform a matrix-vector multiplication with one processing unit and a dot product with another processing unit at the same time. Doing these operations in parallel can reduce the computational latency of the conjugate gradient method, for example [29, pp. 520–527].

Consuming up to 150 W [30] of power, multi-core CPUs can be used for real-time optimization when power consumption is not a primary concern. Compared to microcontrollers, CPUs have more sophisticated computational logic. For this reason, multi-core CPUs can process real numbers in both single and double floating point format, while most microcontrollers cannot. Also, compared to microcontrollers, multi-core CPUs have a higher clock frequency and can do more floating-point operations per second.

3) Graphics Processing Unit (GPU): GPUs are computing devices that consist of eight to sixteen special processors, called multi-threaded processing units (mPU), and a global memory that is shared between the multi-threaded processing units. Each mPU can execute a single instruction on many data sets.

For example, the high-range GPU Nvidia Tesla GK210 [31] has thirteen mPUs, and each one can process 2048 real

TABLE II
COMPARISON BETWEEN PROCESSOR TYPES

Architecture	Parallelism	Power consumption	Clock frequency	Peak FLOPS
Microcontroller	No parallelism in most cases, multiple-instruction-multiple-data in recent architectures	1 mW–1 W	10 MHz–150 MHz	Up to 100 MFLOPS
Multi-core CPU	Multiple-instruction-multiple-data	1 W–150 W	800 MHz–3 GHz	800 MFLOPS–100 GFLOPS
GPU	Single-instruction-multiple-data	1 W–150 W	100 MHz–600 MHz	100 GFLOPS–3 TFLOPS
FPGA	Any	1 W–10 W	10 MHz–350 MHz	Up to 10 TFLOPS

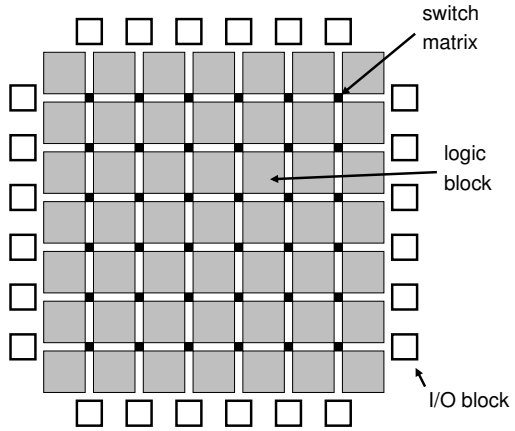

Fig. 8. FPGA architecture

numbers in double precision at the same time. This is almost two orders of magnitude more floating point operations per second compared to the most high performance multi-core CPU.

However, few parallelisable algorithms achieve this performance when carried out on GPUs. Algorithms for dense linear algebra operations, such as dense matrix-vector multiplication, are the best performing algorithms on GPUs. On the other hand, algorithms that execute many conditional statements, such as Givens' QR decomposition, have the worst performance on GPUs.

The average power consumption of a GPU is about the same of that of a multi-core CPU, but the average clock frequency is in the range of hundreds of MHz.

4) Field Programmable Gate Array (FPGA): An FPGA is an integrated circuit that can be configured by a designer with respect to the intended application. At the architecture level, an FPGA represents a matrix of relatively simple logic blocks connected via programmable switches (Figure 8). The two major FPGA vendors, Xilinx and Altera, call these logic blocks Configurable Logic Blocks (CLBs) and Logic Array Blocks (LABs), respectively [32]. Although theoretically any circuit can be implemented using standard logic blocks, contemporary FPGAs have a set of special purpose resources on the chip. This might include DSP slices for high-speed arithmetic operations or RAM blocks for dense data storage [33]. Interaction with the physical world is performed via input/output blocks (IOBs), which are placed around the perimeter of the chip.

The key outstanding feature of FPGAs is customizability. This applies to the data processing unit, memory sub-system and number representation. FPGAs allow implementing only the computational units required by a specific algorithm, unlike CPUs that have fixed logic for performing predefined set of operations. Another important benefit of using FPGAs is memory flexibility. Data can be partitioned into separate memory blocks (ROM/RAM) and placed near the corresponding processing units. This provides huge potential for parallelization of computations both through pipelining and loop unrolling. Furthermore, since both memory word size and computational units are flexible, number representation also becomes variable.

However, extended flexibility comes at the price of some limitations. The first one is reduced clock rate compared to CPU-like architectures. Expanding this bottleneck requires achieving sufficient levels of parallelization, either by implementing deep pipelines or by choosing appropriate loop unrolling factors. Another restriction of an FPGA as an embedded platform is high power consumption in comparison with microcontrollers — this results in additional requirements in terms of the power supply and cooling. Nonetheless, recent research shows some promise in improving energy efficiency of reconfigurable platforms for autonomous applications [34].

Exploiting parallelizability and customizability of FPGAs allows significant reduction of algorithm execution delay and hence sampling time of the system. Further reduction of the sampling time can be achieved by incorporating so-called intra-delay sampling techniques, which imply sampling faster than the computational delay/latency. Intra-delay sampling can be implemented by physically replicating the solver and/or by using data pipelining, so that computation of a new solution is started before termination of previous one. Disturbance rejection capabilities of a controller can be improved as a result of the increased sampling rate [35].

It should be emphasized that the application scope of FPGAs is not limited to target computational platforms. Engineers and researchers extensively use reconfigurable platforms for prototyping to evaluate the functional correctness of hardware designs. FPGAs can be used for fast design exploration by allowing the estimation of silicon and energy usage as a function of design parameters. As a result, the development cost and time-to-market of non-reconfigurable platforms are significantly reduced.

5) Programmable Logic Controller (PLC): A PLC is an industrial computing device intended for automatic process

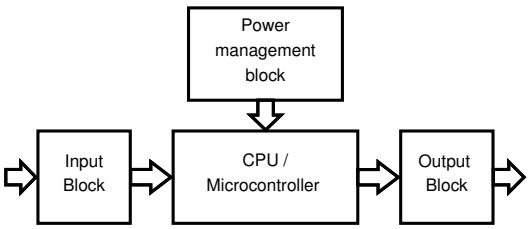

Fig. 9. PLC architecture

control. PLCs are capable of directly interfacing with industrial equipment, including sensors and actuators. This is achieved by using specific input and output modules (Figure 9), which can be chosen and configured with respect to a given digital or analog interface. Power supply units are usually integrated into the device.

PLCs have a set of distinguishing features compared to microcontrollers. Firstly, PLCs are programmed using graphical ladder logic languages, which resemble relay systems that were used before PLCs came into use. This differs from microcontroller programming both in high (C/C++) and low (assembler) level languages. Secondly, PLCs are manufactured as complete devices, which can be used in industry straightaway, whereas a microcontroller is a single chip coming as a part of electronic circuit. In practice, microcontrollers are commonly used as processing units for PLCs. Finally, PLCs can operate with high DC/AC voltages, while typical microcontrollers are compatible only with TTL and/or CMOS technologies [24], [36].

Depending on the computational requirements, a microcontroller or CPU can be used as a processing core in a PLC. Low-cost microcontrollers are ideal for performing logic operations, but not for fast computations, which is crucial for online optimization. However, expressing the solution of an optimization problem as a piecewise linear function of a measured value allows implementing optimization-based algorithms with high sampling rates on low cost PLCs. Nevertheless, large and medium scale optimization problems still require direct or iterative solvers, which are often implemented on relatively expensive CPU-based PLCs capable of performing floating point operations, e.g. [37].

B. Software

Programming the hardware, thus translating the mathematical formulation of the optimization algorithm into software, can be done using a variety of languages. They range from text-based programming languages to graphical ones (e.g. Simulink and LabView) and from generic languages (e.g. C/C++, OpenCL, Python) to domain-hardware specific ones (e.g. DSL, CUDA, HDL).

1) Domain-Specific Languages (DSL): This is a programming approach specifically tailored for a small set of problems belonging to a particular domain [38]. In many cases, a DSL can be seen as a problem specific interface to common sub-routine libraries. DSLs are used in many fields and hundreds of them are available. Their domains range from economics to physics through to creativity and computing. An example in the field of optimization is ZIMPL [39], which is an algebraic modeling language that requires the user to describe a mathematical model in terms of sets depending on parameters. This description is automatically translated into a linear or nonlinear mixed-integer mathematical program that can be fed into a mixed-integer program solver.

2) High level: These are programming languages with a strong level of abstraction in which hardware details are masked to the user. Among these, Python [40] is gaining interest for its ease of use (few lines of code to express complex concepts), code readability and cross-platform compatibility. For these reasons, Python is also largely used in mathematics, science and engineering through the SciPy software [41]. SciPy is a collection of tools and libraries with efficient numerical integration and optimization routines.

3) Model-based: Model-based languages provide the designer a high level of abstraction from implementation details allowing full concentration on algorithm structure. The programming tool can be provided either by a hardware vendor (Xilinx System Generator, Altera DSP builder) or by third parties (Mathworks HDL coder, National Instruments FPGA module). Despite hiding the most low level implementation details, a model-based approach normally supports parallelism, resource usage and energy consumption customization depending on certain design constraints.

Automatic test-bench and stimuli file generation (e.g. [42], [43]) simplifies verification of low level code produced by the compiler. However, the resulting code often suffers from lack of efficiency and readability. This puts some restrictions on the application scope of model-based tools, especially in relation to real-time systems. For this reason, tightly time/resource-constrained applications may still require low level handwritten code [33].

4) C/C++: C [44] and C++ [45] are two of the most popular programming languages among real-time software developers. The success of these two programming languages is due to the high speed of the compiled code and the availability of compilers and development tools for many processors. Software libraries can be used inside C and C++ source code to help in the development of real-time optimization software. Eigen [46] is a C++ library for both dense and sparse linear algebra. Eigen contains several linear solvers and matrix decomposition algorithms, but carries out nonlinear optimization as well. Independently from the processor type, the source code of Eigen can be compiled with any C++ compiler. The GNU scientific library (GSL) [47] can be used with both C and C++ when the processor is a CPU. GSL includes not only linear algebra algorithms, but also simulated annealing and least-squares fitting. Intel MKL [48] is a software library that is optimised for Intel CPUs, but its source code is not public.

5) Vendor-specific: Some microcontrollers and PLCs cannot be programmed in a general purpose programming language like C and C++. In this case, a language specifically designed by the vendor is used. Sofware written in vendor-specific programming languages are highly optimised for specific hardware architectures. This software cannot be

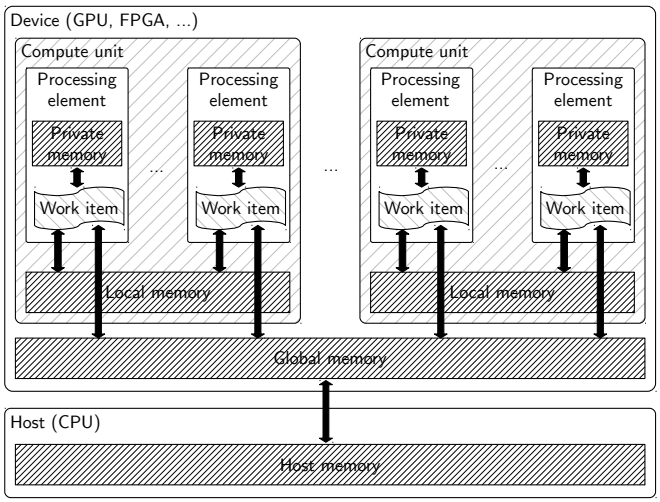

Fig. 10. OpenCL model of the architecture of a parallel accelerator.

executed on other hardware architectures or vendors. For example, the Arduino language [49] is only used to program the microcontroller in Arduino boards. CUDA [50] is a popular vendor-specific programming language in the field of scientific computing. The syntax of CUDA is similar to C and C++, but CUDA can only be used to program Nvidia GPUs.

6) Open Computing Language (OpenCL): Recently, both industry and academia have been interested in standardising how to program any kind of hardware accelerator, independently from its architecture. Figure 10 illustrates the OpenCL model [51].

In this model, a device is composed of a number of processing elements (PEs), which are the most basic computational blocks in the architecture. A PE executes an elementary piece of software, called a work item, and contains a small private memory. PEs are packed into compute units (CUs) and share a local memory. This has a higher access latency compared to the private memory of each PE, but is also larger.

The set of all work items being processed by a compute unit is called a work group. A device can have many compute units, all sharing a global memory and each one processing a different work group. Work groups are executed out-of-order, which means their computation has to be mutually independent. In fact, the execution of work items in a work group can be synchronised, but that of work items in different work groups cannot.

The set of all work groups processed on a device is called a kernel. A device can process more work groups than its compute units, because at the end of the execution of a work group the local memory is flushed and a new work group is loaded. For this reason, any data in common between different kernels has to be stored in the global memory.

Table III illustrates how the model discussed above maps onto the actual hardware of three different types of parallel devices: GPUs, multi-core CPUs and FPGAs. A discussion of the FPGA case can be found also in [52], [53].

7) Hardware Description Language (HDL): An HDL is a formal language for describing the architecture of an integrated circuit. In contrast to software programming languages that define a set operations to be performed by a processor (control flow), HDLs describe the architecture of the processor itself (data flow). VHDL and Verilog are the most widely used HDLs.

In practice, depending on the target platform, HDL code is usually mapped to an existing array of logic gates, as for FPGAs, or implemented using standard library cells, as for Application Specific Integrated Circuits (ASICs)) [32]. An HDL is the most efficient FPGA configuration approach that, however, suffers from implementation complexity and hence has a high entry barrier for designers.

The process of building an FPGA circuit from its HDL description involves *synthesis* and *place-and-route*. Synthesis is essentially constructing a connected graph, called a *netlist*, that describes circuit logic gates and their interconnection. Place-and-route, which follows synthesis, involves solving a series of optimization problems in order to fit the netlist into the physical device. The resulting configuration data can be uploaded to an FPGA in the form of a *bitstream*.

8) Design Tools: Using dedicated design tools for real-time optimization significantly speeds up the development process both from the code generation and code implementation perspectives. Code generation tools accept a high level description of an optimization problem to generate reliable custom code (mainly C/C++) for solving that particular problem. The resulting code is often not fixed to any platform and can be compiled for various real-time embedded processors. Contemporary generation tools avoid using third party libraries and dynamic memory allocation to ensure robustness of the solver. The majority of code generation tools exploit the structure of optimization problem and precomputes all time-invariant values in order to satisfy tight timing constraints. The tool might be intended for optimization in general (CVXGEN) or suited to problems arising in particular areas, e.g. in optimal control or estimation (ACADO, qpOASES, FORCES).

VI. Future Architectures

This paper has outlined some of the issues that arise with existing computer architectures, especially parallel architectures for high performance embedded systems. However, it is worth looking to the future to understand the trends of computer architecture and how they may impact on real-time optimization applications.

The move to parallelism for high performance, even in embedded systems, has been driven largely by power consumption considerations: for the same performance, it has become far more efficient to implement many simple processors operating in parallel than one complex low-latency processor. However, there is a limit to efficiency gains that come from adopting simpler von Neumann architectures, leading to the need to fundamentally rethink microarchitecture [54]. We envisage a number of trends in future

TABLE III
CORRESPONDENCE BETWEEN THE OPENCL ABSTRACT MODEL [51] AND ACTUAL ACCELERATOR ARCHITECTURES

Concept	GPU	Multi-core CPU	FPGA
Processing element (PE)	A SIMD lane. Each SIMD lane has up to 8 arithmetic-logic units (ALUs). Can perform the same operation on many integer or single-precision floating point numbers at the same time.	Each core is composed of a single PE, which usually supports a limited vectorisation of operations.	Defined ad-hoc for a given kernel.
Private memory	A small DRAM memory private to each SIMD lane. The size is in the order of tens of bytes.	Banks of registers inside each core.	Configurable by the designer, usually based on flip-flops.
Compute unit (CU)	A multithreaded processing unit (mPU). Composed of up to 32 lanes or processing elements.	A processing unit.	Ad-hoc parallel architecture for the given kernel.
Local memory	Fast SRAM memory whose size is in the order of kilobytes. An average GPU has 32 KB local memories. A local memory is available to each CU.	Cache memory. An average mCPU has 32 KB local memories.	Block RAM (BRAM). The size is configurable by the designer.
Global memory	DRAM memory accessible to all multithreaded SIMD processors in a GPU.	A region of the RAM memory on the computer's motherboard.	Configurable by the designer. Either off-chip RAM in the FPGA board or BRAM.

architectures that will affect the methods used for real-time optimization:

- There will be more heterogeneous hardware, incorporating special purpose functional units, in order to avoid power-inefficiencies of general purpose computing. This means that further work will be required to develop compilation frameworks from the specification of the real-time optimization problem down to the implementation in heterogeneous hardware. These frameworks will need to be parameterisable by the available special purpose units, to avoid the human effort that would be required to program these architectures using today's technologies. It is likely that one or more "embedded control" system-on-chip devices with custom units developed specifically for real-time optimization will become available, and the user community will need to work with vendors to help specify these devices.
- There will be the need for algorithms to tolerate hardware unreliability. The illusion of total hardware reliability has always been maintained at a price in power consumption and performance. In small geometry CMOS processes, this price is increasing rapidly [55]. Real-time optimization provides an interesting use case for architectures whereby total correctness need not be preserved at the level of a single optimization solve, so long as the global system behaviour is preserved. Thus the embedded control SoCs we may see emerge in the future are likely to have quite different error-recovery mechanisms compared to those used in the data centre, and these need to be formalised and integrated into the design process of both the control algorithm and the hardware specification.

VII. CONCLUSIONS

The efficiency of a real-time optimization algorithm is a function of the computational hardware on which it has been implemented. Furthermore, since the computing system is affected by measurements from a physical system or process that is evolving in time, it is essential to consider for what purpose the solution to the sequence of optimization problems will be used. While the computation is being carried out, the system is subject to uncertainties. Hence, the correctness of the implementation should be assessed based on the performance of the cyber-physical system, rather than just considering the algorithm on its own. It is not necessary, for example, to use double precision floating point arithmetic and compute a globally optimal point to have acceptable performance. An inaccurate and imprecise solution implemented at a fast rate might be preferable to an accurate and precise solution implemented at a slow rate.

There is currently a wide range of processors available that allow the designer to trade off the time, energy, space and cost of the computer system against the performance and robustness of the overall system. There has also been considerable developments in recent years in developing frameworks, languages and software tools that make the implementation easier and enable the use of a variety of targets suitable for real-time use.

We therefore envisage a future of heterogenous computing platforms made up of general-purpose, highly parallelizable, application-specific and reconfigurable architectures. Some of these architectures will have been specifically designed for the kinds of mathematical operations needed in real-time optimization. Engineers will therefore not have to deliver sub-optimal implementations of an optimization algorithm onto an inflexible computing system. Instead, advanced design methods and tools will allow them to mix and match hardware

and software components suitable for the application.

ACKNOWLEDGEMENTS

The authors would like to acknowledge funding from EC FP7 grant 607957 (TEMPO), EPSRC grants EP/I020357/1 and EP/I012036/1, Imagination Technologies and the Royal Academy of Engineering.

REFERENCES

[1] C. Rao, J. Rawlings, and D. Mayne, "Constrained state estimation for nonlinear discrete-time systems: stability and moving horizon approximations," *Automatic Control, IEEE Transactions on*, vol. 48, no. 2, pp. 246–258, Feb 2003.

[2] J. B. Rawlings and D. Q. Mayne, *Model Predictive Control: Theory and Design*. Nob Hill Publishing, 2009.

[3] J. Humpherys, P. Redd, and J. West, "A fresh look at the kalman filter," *SIAM Review*, vol. 54, no. 4, pp. 801–823, 2012. [Online]. Available: http://dx.doi.org/10.1137/100799666

[4] D. Q. Mayne, "Model predictive control: Recent developments and future promise," *Automatica*, vol. 50, no. 12, pp. 2967 – 2986, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0005109814005160

[5] L. T. Biegler, *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. SIAM, 2010. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9780898719383

[6] J. T. Betts, *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, 2nd ed. SIAM, 2010.

[7] T. A. Davis, *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[8] A. Greenbaum, *Iterative Methods for Solving Linear Systems*. SIAM, 1997.

[9] J. Kang, N. Chiang, C. D. Laird, and V. M. Zavala, "Nonlinear programming strategies on high-performance computers," in *Proc. 54th IEEE Conference on Decision and Control*, Osaka, Japan, 2015.

[10] J. Nocedal and S. Wright, *Numerical Optimization*, Second, Ed. Springer, 2006.

[11] V. Cevher, S. Becker, and M. Schmidt, "Convex optimization for big data," *IEEE Signal Processing Magazine*, September 2014.

[12] A. Shahzad, E. C. Kerrigan, and G. A. Constantinides, "A stable and efficient method for solving a convex quadratic program with application to optimal control," *SIAM Journal on Optimization*, vol. 22, no. 4, pp. 1369–1393, 2012. [Online]. Available: http://dx.doi.org/10.1137/11082960X

[13] G. A. Constantinides, "Tutorial paper: Parallel architectures for model predictive control," *Proc. European control conference*, pp. 138–143, 2009.

[14] *LogiCORE IP. Floating Point Operator. Product Guide*, 7th ed., Xilinx, April 2014.

[15] J. L. Jerez, S. Richter, P. J. Goulart, G. A. Constantinides, E. C. Kerrigan, and M. Morari, "Embedded online optimization for model predictive control at megahertz rates," *Automatic Control, IEEE Transactions on*, vol. 59, no. 12, pp. 3238–3251, Dec 2014.

[16] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. Hemmert, "A comparison of floating point and logarithmic number systems for FPGAs," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, April 2005, pp. 181–190.

[17] P. Vouzis, M. Kothare, L. Bleris, and M. Arnold, "A system-on-a-chip implementation for embedded real-time model predictive control," *Control Systems Technology, IEEE Transactions on*, vol. 17, no. 5, pp. 1006–1017, Sept 2009.

[18] C. T. Ewe, "Dual fixed-point: an efficient alternative to floating-point computation for DSP applications," in *Field Programmable Logic and Applications, 2005. International Conference on*, Aug 2005, pp. 715–716.

[19] E. C. Kerrigan, "Co-design of hardware and algorithms for real-time optimization," in *Control Conference (ECC), 2014 European*, June 2014, pp. 2484–2489.

[20] A. Messac and C. Mattson, "Normal Constraint Method with Guarantee of Even Representation of Complete Pareto Frontier," *AIAA Journal*, vol. 42, pp. 2101–2111, Oct. 2004. [Online]. Available: http://dx.doi.org/10.2514/1.8977

[21] G. Kirlik and S. Sayn, "A new algorithm for generating all non-dominated solutions of multiobjective discrete optimization problems," *European J. Operational Research*, vol. 232, no. 3, pp. 479–488, 2014.

[22] A. Custodio, M. Emmerich, and J. Madeira, "Recent developments in derivative-free multiobjective optimisation," *Computational Technology Reviews*, vol. 5, pp. 1 – 30, 2012.

[23] M. Tabatabaei, J. Hakanen, M. Hartikainen, K. Miettinen, and K. Sindhya, "A survey on handling computationally expensive multiobjective optimization problems using surrogates: non-nature inspired methods," *Structural and Multidisciplinary Optimization*, vol. 52, no. 1, pp. 1–25, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00158-015-1226-z

[24] *8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash – ATmega48PA ATmega88PA ATmega168PA ATmega328P*, Rev. 8161 ed., ATMEL, Oct. 2009. [Online]. Available: www.atmel.com/images/8161s.pdf

[25] *Cortex-A7 MPCore Technical Reference Manual*, r0p3 ed., ARM, May 2012. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/index.html

[26] C.-K. Chui, B. P. Nguyen, Y. Ho, Z. Wu, M. Nguyen, G.-S. Hong, D. Mok, S. Sun, and S. Chang, "Embedded real-time model predictive control for glucose regulation," in *Proceedings of the World Congress on Medical Physics and Biomedical Engineering*, ser. IFMBE Proceedings, M. Long, Ed., vol. 39. Springer Berlin Heidelberg, 2013, pp. 1437–1440.

[27] *C2000 Real-Time Microcontrollers*, Texas Instruments, 2015. [Online]. Available: http://www.ti.com/lit/sg/sprb176x/sprb176x.pdf

[28] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C–21, no. 9, pp. 948–960, Sep. 1972.

[29] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[30] "Intel core i7-3970x processor extreme edition specifications," Online, 2012. [Online]. Available: http://ark.intel.com/products/70845/Intel-Core-i7-3970X-Processor-Extreme-Edition-15M-Cache-up-to-4_00-GHz

[31] *Tesla K80 GPU Accelerator*, Nvidia, Jan. 2015. [Online]. Available: http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf

[32] V. Pedroni, *Circuit Design with VHDL*. MIT Press, 2004. [Online]. Available: https://books.google.com/books?id=b5NEgENaEn4C

[33] L. Crockett, R. Elliot, and M. Enderwitz, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014. [Online]. Available: https://books.google.com/books?id=9dfvoAEACAAJ

[34] P. Jamieson, W. Luk, S. Wilton, and G. Constantinides, "An energy and power consumption analysis of FPGA routing architectures," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Dec 2009, pp. 324–327.

[35] D. Buchstaller, E. Kerrigan, and G. Constantinides, "Sampling and controlling faster than the computational delay," *IET Control Theory and Applications*, vol. 6, pp. 1071–1079, 2012. [Online]. Available: http://dx.doi.org/10.1049/iet-cta.2010.0440

[36] *STM32F303xB STM32F303xC product data*, 11st ed., STMicroelectronics, Apr. 2015. [Online]. Available: http://www.st.com/web/en/resource/technical/document/datasheet/DM00058181.pdf

[37] *S7-200 Programmable Controller System Manual*, Siemens, Postfach 4848, D-90327 Nuernberg, September 2007.

[38] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[39] T. Koch, "Rapid mathematical prototyping," Ph.D. dissertation, Technische Universität Berlin, 2004.

[40] *Python*, 2015. [Online]. Available: http://www.python.org

[41] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 2015-06-10]. [Online]. Available: http://www.scipy.org/

[42] *Vivado Design Suite User Guide. Model-Based DSP Design using System Generator*, Xilinx, April 2014.

[43] *DSP Builder Handbook*, 14th ed., Altera, December 2014.

[44] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[45] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[46] G. Gaël, J. Benoît *et al.*, "Eigen v3," Online, 2010, accessed May 28th, 2015. [Online]. Available: http://eigen.tuxfamily.org

[47] B. Gough, *GNU Scientific Library Reference Manual*, 3rd ed. Network Theory Ltd., 2009.

[48] F. Jeanette, *Intel Math Kernel Library Reference Manual*, 11st ed., Intel, Aug. 2014. [Online]. Available: software.intel.com/en-us/mkl_11.2_ref_pdf

[49] B. W. Evans, *Arduino programming notebook*, 2007. [Online]. Available: playground.arduino.cc/uploads/Main/arduino_notebookv1-1.pdf

[50] *CUDA Toolkit Documentation*, 5th ed., Nvidia corporation, Oct. 2013, accessed 1 March 2014. [Online]. Available: http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html

[51] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[52] *Implementing FPGA Design with the OpenCL Standard*, Nov. 2013. [Online]. Available: http://www.altera.co.uk/literature/wp/wp-01173-opencl.pdf

[53] *The Xilinx SDAccel Development Environment*, 2014. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-backgrounder.pdf

[54] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Berger, "Dark silicon and the end of multicore scaling," in *Proc. 38th International Symposium on Computer Architecture (ISCA'11)*, 2011.

[55] D. Ernst, N.-S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.