Software and Hardware Code Generation for Predictive Control Using Splitting Methods [⋆]

Harsh A. Shukla [*] **Bulat Khusainov** [**] **Eric C. Kerrigan** [***] **Colin N. Jones** [*]

[*] *Laboratoire d'Automatique, Ecole Polytechnique Fédérale de Lausanne, Switzerland. (e-mail: harsh.shukla, colin.jones@epfl.ch).*
[**] *Department of Electrical and Electronic Engineering, Imperial College London, London, SW7 2AZ, UK (e-mail: b.khusainov@imperial.ac.uk)*
[***] *Department of Electrical & Electronic Engineering and Department of Aeronautics, Imperial College London, London, SW7 2AZ, UK (e-mail: e.kerrigan@imperial.ac.uk).*

Abstract:
This paper presents SPLIT, a C code generation tool for Model Predictive Control (MPC) based on operator splitting methods. In contrast to existing code generation packages, SPLIT is capable of generating both software and hardware-oriented C code to allow quick prototyping of optimization algorithms on conventional CPUs and field-programmable gate arrays (FPGAs). A Matlab interface is provided for compatibility with existing commercial and open-source software packages. A numerical study compares software, hardware and heterogeneous implementations of splitting methods and investigates MPC design trade-offs. For the considered testcases the reported speedup of hardware implementations over software realizations is 3x to 11x.

Keywords: Model predictive and optimization-based control; Hardware-software co-design; Embedded computer architectures; Hardware-in-the-loop simulation, Splitting methods.

1. INTRODUCTION

Model Predictive Control (MPC) is a control technique that aims to minimize a predefined cost function satisfying the design constraints. The cost function often reflects the difference between predicted and desired trajectories of the system states and inputs, while constraints capture physical limitations of the system. Thus an MPC controller requires the solution of an optimization problem at every sampling instant, leading to the following challenges that prevents further expansion of MPC:

- Solving an optimization problem efficiently so that one can compute the control law within a sampling interval.
- Quickly prototyping and deploying the controller on different types of embedded systems.
- Trading off computational resources for FPGAs against closed-loop performance of the resulting cyber-physical system.

Thanks to progress on hardware development and efforts on numerical computing, there are various toolboxes available to address the first two challenges. However, in this work we address all the above-mentioned challenges by proposing an open source and free toolbox, namely SPLIT. SPLIT generates code in C and the user interface is written in high-level language (MATLAB). The idea is to provide an easy to use toolbox, which allows rapid prototyping and deployment of an MPC-based controller on embedded processors, FPGAs or heterogeneous platforms. To solve an optimization problem created from an MPC-based controller, the toolbox uses operator splitting methods, also known as decomposition methods.

The key feature of operator splitting methods is to decompose the original optimization problem into subproblems, which are computationally cheaper to solve than the original problem. Operator splitting methods have enjoyed popularity in large scale optimization problems for a long time and are gaining attention for small to medium scale embedded optimization problems. Using SPLIT, users can choose from a family of operator splitting methods, and quickly define the optimization problem in MATLAB (roughly 5–10 lines of code). SPLIT generates corresponding C code, which can be deployed readily on embedded processors. Another important feature of SPLIT is support of code generation for FPGAs as well, due to the use of the Protoip toolbox. The Protoip toolbox enables users to go smoothly from C code to deployment on FPGAs. Without having any prior knowledge about FPGAs, users can directly implement the designed controller.

[⋆] This work has received funding from the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement no 607957 (TEMPO).

Algorithm 1 Alternating Minimisation Algorithm (AMA)

```
Step 1: $z^{k+1} = \underset{z}{\text{argmin}} \quad f(z^k) + \langle \lambda^k, Lz^k + l - y^k \rangle$
Step 2: $y^{k+1} = \underset{y}{\text{argmin}} \quad g(y^k) + \langle \lambda^k, Lz^{k+1} + l - y^k \rangle$
            $+(\rho/2)\|Lz^{k+1} + l - y^k\|^2$
Step 3: $\lambda^{k+1} = \lambda^k + \rho(Lz^{k+1} + l - y^{k+1})$
```

The organization of the paper is as follows. We briefly introduce splitting methods in Section 2. In Section 3, we provide a summary for different features of SPLIT. We introduce reconfigurable and heterogeneous computing in Section 4. In Section 5 we discuss details on how SPLIT efficiently tailors the algorithm for reconfigurable and heterogeneous platform and deploys generated code using Protoip toolbox. Numerical examples are provided in Section 6. Finally, Section 7 concludes the paper.

2. SPLITTING ALGORITHMS

In this section we discuss how splitting algorithms can be applied to solve predictive control problems. We will particularly focus on numerical operations involved in splitting algorithms. This section is based on Stathopoulos et al. (2016). Consider the following optimization problem:

$$\begin{array}{ll} \text{minimize} & f(z) + g(y) \\ \text{subject to} & Lz - y = -l \end{array} , \tag{1}$$

with variables $z \in \mathbb{R}^n$, $y \in \mathbb{R}^m$ where $f : \mathbb{R}^n \to (-\infty, \infty]$ and $g : \mathbb{R}^m \to (-\infty, \infty]$ are proper, lower semi-continuous convex functions, $l \in \mathbb{R}^m$ and $L \in \mathbb{R}^{m\times n}$.

We want to solve an optimization problem of the form (1) using splitting methods ,which can be seen as a three-step algorithm. In the first two steps minimization with respect to variables z and y is performed, followed by the third step to bring consensus by the updating dual, corresponding to equality constraints. The key feature of splitting methods is that each of the three steps is computationally cheap and often has a closed form solution. As an example, in Algorithm 1, we illustrate the three steps for a particular splitting method, namely the Alternating Minimization Algorithm.

It is easy to rewrite a linear MPC problem in the form of (1) and use splitting methods to solve the resulting optimization problem. In such cases, f in (1) is a quadratic objective function defined over a linear system and g is an indicator function on a set (e.g. positive orthant). We then see that the three steps in AMA comes down to the following three numerical operations:

- Solving a system of linear equations;
- Matrix-vector multiplications;
- Vector-vector manipulations.

The most computational costly operation is solving a linear system of equations. It is often the case that splitting methods give rise to a KKT (Karush-Kuhn-Tucker) system of the form

$$\begin{bmatrix} K_{11} & K_{21}^\top \\ K_{21} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{2}$$

Fortunately, for splitting methods, the KKT matrix does not change over iterations (except very few anomaly cases) and thus one can precompute the inverse or matrix factorization (e.g. LDL^T). It is important to note that matrices involved are often sparse and structured. Thus, one can exploit this fact while either solving the linear system of equations or computing matrix-vector multiplications. In the following section, we introduce the SPLIT toolbox and discuss how SPLIT utilizes sparsity arising from the splitting methods for efficient computation on embedded processors.

3. CODE GENERATION FOR SOFTWARE USING SPLIT

The purpose of SPLIT is to provide an easy-to-use open source and free toolbox that saves time and effort for users deploying splitting methods on embedded systems. The toolbox is written in a high-level language (MATLAB) and code generation is tailored to a specific splitting algorithm. In this way the toolbox exploits the problem structure for generating efficient C code. SPLIT can be used to target an embedded general-purpose processor or it can also be used to deploy on pure FPGAs or heterogeneous platforms. In this section, we will focus particularly on deployment on general-purpose embedded processors and Section 5 is dedicated for detailed discussions on code generation for FPGAs.

The toolbox supports three splitting methods:

- Alternating Direction Method of Multipliers (Gabay and Mercier, 1976);
- Alternating Minimization Algorithm (Tseng, 1991);
- Primal Dual Algorithm which is also known as Vũ-Condat Algorithm (Condat, 2013).

It is also possible to enable different features for the listed algorithms: acceleration based on Nesterov's relaxation, preconditioning and adaptive restart. We refer to Stathopoulos et al. (2016) for details.

Since SPLIT has its own libraries it can be used as a library-free toolbox. The toolbox also provides users an option to select different linear algebra libraries like SuiteSparse (Davis and Hu, 2011), BLAS (Lawson et al., 1979) and LAPACK (Anderson et al., 1999). Thus, depending on an application, a user can use SPLIT as library-free or with a suitable library. Another strength of the toolbox is that it analyses the sparsity pattern of the problem and recommends to the user a particular library or method.

As discussed in Section 2, the computationally expensive operations for splitting algorithms are solving linear systems and matrix-vector multiplication. We summarize various ways to solve linear systems and computing matrix vector multiplications using SPLIT in Table 1 and Table 2, respectively.

- The idea behind the Custom method in Table 1 is to compute the $LDL^\top$ factorization in MATLAB, and then explicitly write the entries of the factorization in a generated C file.

Table 1. Different linear system solution methods supported by SPLIT

Library	Sparsity	Method	Suitable for
Custom	Sparse	LDL	Small sparse matrices
SuiteSparse	Sparse	LDL	Large sparse matrices
CLAPACK	Dense	LDL	Dense matrices

Table 2. Matrix-vector multiplications supported by SPLIT

Library	Suitable for
BLAS	Dense matrices
SuiteSparse	Sparse matrices
Custom_software	Small sparse matrices
Custom_HW_sparse	Sparse matrices on FPGAs
Custom_HW_dense	Dense matrices on FPGAs

- It is important to note that for solving a linear system based on precomputing the inverse offline and performing matrix -vector multiplication online, any method provided in Table 2 can be used.
- The idea for the Custom_software method in Table 2 is to explicitly write the entries of the matrix in a generated C file.
- Custom_HW_sparse and Custom_HW_dense are tailored for implementation on FPGAs and they are explained in Section 5.

Before we discuss how SPLIT deploys splitting algorithms on FPGAs, we first introduce reconfigurable and heterogeneous computing in the following section.

4. RECONFIGURABLE AND HETEROGENEOUS COMPUTING PLATFORMS

An FPGA is an array of relatively simple circuits, namely flip-flops (FFs) and lookup tables (LUTs), that are connected to switch matrices. Configuring switch matrices allows creating connections between and within logic blocks so that the desired circuit is obtained. Modern FPGAs provide special purpose units (e.g. dedicated memory blocks) that are more efficient from a silicon usage and signal routing point of view, compared to general purpose FFs and LUTs (Crockett et al., 2014).

The key outstanding feature of reconfigurable platforms is customizability. Unlike fixed architecture CPUs that have fixed logic for performing a predefined set of operations, FPGAs allow synthesizing computational units with respect to a given algorithm. Moreover, computational units can be connected directly to each other to create *data pipelines*. The data storage subsystem is also flexible; memory blocks can be partitioned and placed near the corresponding processing units so that each block is processed independently.

However, the above-mentioned advantages often come at the price of certain limitations. Firstly, FPGA clock rates are often slower compared to CPU-like architectures. Overcoming this limitation requires introducing a sufficient degree of parallelism. Secondly, some algorithms cannot be efficiently mapped on hardware due to data dependencies or resource-consuming operations. In such cases it might be useful to employ heterogeneous platforms, known as a systems-on-a-chip (SoCs), that incorporate both conventional CPUs and FPGAs. With a heterogeneous computing approach, the computationally heavy part of the workload can be accelerated on the FPGA, while keeping the rest of the algorithm on the CPU to save computational resources. The SPLIT toolbox considers three options of splitting the workload:

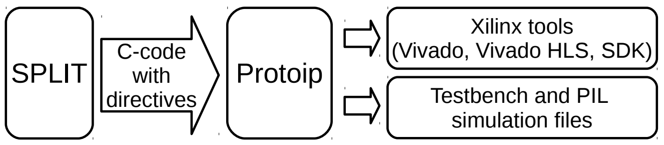
SPLIT
C-code with directives
Protoip
Xilinx tools (Vivado, Vivado HLS, SDK)
Testbench and PIL simulation files

Fig. 1. The proposed toolchain flow.

- Pure software implementation;
- Heterogeneous implementation: accelerating the linear system solver on the FPGA and computing the rest on the CPU;
- Pure FPGA implementation.

A heterogeneous computing platform can be programmed in several ways. Conventional approaches propose programming each subsystem using a dedicated, often low-level language and handling data transfer between subsystems manually. Although low-level programming often leads to efficient realizations both from a time and resource usage point of view, these benefits come at the price of high implementation effort and hence long time-to-market. Model-based languages (e.g. the Matlab HDL coder) on the other hand significantly reduce implementation effort, providing flexibility of shifting the workload between different computational subsystems and allowing quick closed-loop performance verification. Unfortunately, the resulting circuit often cannot be considered as efficient.

The use of C-based integrated development environments (e.g. Xilinx SDSoC) is a compromise between design effort and implementation efficiency: although the whole computing system is programmed using a unified language, supplying the code with additional compiler directives allows specifying low level details to satisfy given design constraints. The SPLIT toolbox relies on a C-based approach and uses the Protoip tool (Suardi et al., 2015) to manage underlying projects (see Figure 1). Protoip allows rapid prototyping of optimization algorithms on FPGAs by providing processor-in-the-loop (PIL) test facilities while abstracting low-level implementation details. PIL simulation allows verifying controller performance by running optimization algorithm on FPGA and simulating the plant on a desktop machine.

5. CODE GENERATION FOR HARDWARE USING SPLIT

We proceed to explain how SPLIT allows users to directly deploy an MPC controller on FPGAs or on heterogeneous platforms. The flow for code generation for FPGAs is illustrated in Figure 1.

Following this, SPLIT generates hardware-oriented synthesizable code with synthesis directives (e.g. pipelining, parallelization) that allow efficient mapping on hardware.

```
// SW  oriented  code
float vec_sum_sw(float vec_in[1000])
{
  int i;
  float sum 0;
  for(i = 0; i < 1000; i++)
  {
#pragma HLS PIPELINE
    sum += vec_in[i];
  }
  return sum;
}
// HW oriented code
float vec_sum_hw(float vec_in[1000])
{
  int i,j;
  int mask[2] = {0, ~((int) 0)};
  float sum_p[8] = {0};
  for(i = 0, j = 0; i < 1000; i++) // partial
      accumulation
  {
#pragma HLS DEPENDENCE variable=sum_p array inter
     distance=8 true
#pragma HLS PIPELINE
    sum_p[j] += vec_in[i];
    j = (j+1) & mask[(j+1) != 8];
  }
  for(i = 1; i < 8; i++) // final accumulation
  {
#pragma HLS UNROLL
    sum_p[0] += sum_p[i];
  }
  return sum_p[0];
}
```

Fig. 2. Software- and hardware-oriented C-code for calculating the sum of vector elements.

We classify all underlying computations for splitting algorithms into scalar, vector-vector and matrix-vector operations.

Scalar operations, e.g. computing Nesterov's relaxation, do not require acceleration, since the computational complexity for these operations does not increase with the problem size.

Vector-vector operations often can be accelerated by pipelining the loop that iterates over vectors elements. For the simplest case, when consecutive iterations do not depend on each other (e.g. element-wise addition), this is implemented by supplying the source code with a pipelining directive. However, for some vector operations pipelining cannot be implemented in a straightforward manner, which is a consequence of read-write data dependencies. Consider the example of computing the sum of a vector's elements (see Figure 2). With a software-oriented approach the next iteration of the main loop cannot be started before finishing the previous iteration due to data reading and writing dependencies, which applies restrictions on pipelining. Hardware-oriented code computes partial sums of the vector's elements independently, which allows avoiding undesirable data dependencies and creating an efficient pipeline. After the partial accumulation is finished, the final loop accumulates the partial sums. According to the report from the synthesis tool Vivado HLS, the FPGA implementation of `vec_sum_hw()` is 4.75x faster compared to `vec_sum_sw()`. Note that the same memory access pattern with the considered example holds for calculating vector norms, vector-vector and matrix-vector multiplications, which is exploited by SPLIT.

Matrix-vector computations required for splitting methods can be classified into dense matrix-vector multiplication, sparse matrix-vector multiplication and sparse forward/backward substitution. Dense matrix-vector multiplication essentially represents a set of vector-vector multiplications, which can be computed in parallel. SPLIT allows trading off computation time against FPGA resource usage by changing the degree of parallelism, i.e. the number of rows processed in parallel. Regarding sparse matrix-vector computations, SPLIT handles non-zero elements in a certain order that allows avoiding data dependencies and hence opens the possibility of pipelining. This is achieved by scheduling, i.e. determining the order of processing the non-zero elements, offline during the code generation stage.

Once tailored C code with synthesis directives is generated, SPLIT uses the Protoip toolbox for synthesizing the code and deploying the controller on an FPGA. The use of Protoip on the underlying level allows quick prototyping of the algorithms on hardware as well as closed-loop performance verification with processor-in-the-loop tests. Note that the entire design flow is fully automated and no prior FPGA knowledge is required from a user.

In this way, by defining an MPC-based formulation in a high-level language like MATLAB, users can deploy a controller on FPGAs or heterogeneous platforms. This is the first free and open source toolbox that provides code generation and deployment of an MPC-based controller on FPGAs or heterogeneous platforms.

6. EXPERIMENTAL RESULTS

In this work we use a Xilinx Zynq-7000 XC7Z020 SoC with dual-core ARM Cortex-A9 and FPGA logic, which contains: 53200 LUTs, 106400 FFs, 220 DSP blocks and 140 block RAMs with total capacity 4.9 Mb. A high throughput-oriented AXI bus provides fast communication between the CPU and FPGA and hence allows heterogeneous implementations of computationally intensive algorithms.

We consider the following optimal control problem for the rest of the section:

$$\begin{aligned}
\text{minimize}\ & \sum_{i=0}^{N-1} \left(x_i^\top Q x_i + u_i^\top R u_i\right) + x_N^\top Q_N x_N \\
\text{subject to}\ & x_{i+1} = Ax_i + Bu_i,\ \text{for } i = 0,\dots,N-1\ , \\
& x_i \in \mathcal{X}\ ,\ \text{for } i = 0,\dots,N\ , \\
& u_i \in \mathcal{U}\ ,\ \text{for } i = 0,\dots,N-1\ , \\
& x_0 = \hat{x}\ ,
\end{aligned} \tag{3}$$

where the vector $x_i \in \mathbb{R}^{n_x}$ represents states, $u_i \in \mathbb{R}^{n_u}$ is the input vector, Q and R are penalty matrices on states and inputs with appropriate dimensions. $A \in \mathbb{R}^{n_x \times n_x}$ is the state transition matrix, $B \in R^{n_x \times n_u}$ is the input matrix. The sets $\mathcal{X}$ and $\mathcal{U}$ are convex and define constraints on the states and inputs, respectively. $\hat{x}$ is an initial state and N is a prediction horizon.

6.1 Software, heterogeneous and hardware implementations

In this example, we randomly create predictive control problems of the form (3) while varying the number of states, inputs and horizon length as provided in Table 3.

Table 3. Generated problems with varying size

	n_u	n_x	N	Dimensions
Problem 1	2	4	4	48
Problem 2	4	8	7	156
Problem 3	6	12	12	384

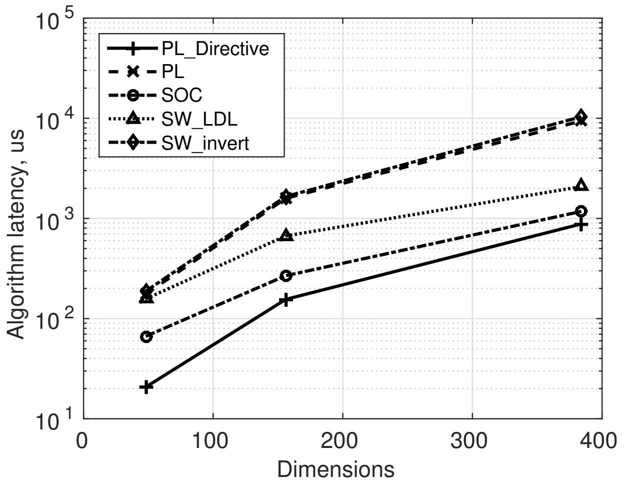

Fig. 3. Time per iteration with varying size of problem on a log scale

The parameter Dimensions in Table 3 is the matrix size when solving a linear system of equations. We incorporate box constraints on inputs. The optimization problem is solved using an accelerated version of AMA with single precision floating point arithmetic.

The goal is to compare and study the latency and resource usage for implementation on an FPGA, heterogeneous platform and on a general-purpose embedded processor. We consider the following five scenarios:

(i) **Implementation on FPGA with synthesis directives:** In this case, all the numerical operations are performed on an FPGA. The code generated by SPLIT with synthesis directive is used. The latency is illustrated in Figure 3 as PL_directive.

(ii) **Implementation on FPGA without synthesis directive:** In this case, all synthesis directives are commented to compare and study efficiency of generated C code with directives by SPLIT. The latency performance is illustrated in Figure 3 as PL.

(iii) **Heterogeneous implementation:** In this experiment, all the operations, except solving linear systems, are computed on the embedded processor and the linear system is solved on an FPGA. This is because the computational bottleneck for splitting methods is the solution of a linear system. Since the matrix involved in solving the linear system does not change over iterations, it is preloaded on the FPGA at circuit synthesis stage. Thus there is no online memory communication for the matrix between the embedded processor and FPGA. Heterogeneous implementation is also inherently supported by SPLIT. The latency of this approach is illustrated in Figure 3 as SOC.

Table 4. DSP usage in %

	Problem 1	Problem 2	Problem 3
PL_directive	11.82	13.64	13.64
PL	11.82	12.73	12.73
SOC	2.27	2.27	2.27

(iv) **Software-based implementation of LDL:** In this approach, all the numerical calculations are performed using the onboard dual-core ARM Cortex-A9 processor. The linear system is solved using LDL factorization. In Figure 3, this is denoted by SW_LDL.

(v) **Software-based implementation of mat-vec:** To compare the performance of LDL factorization, we precompute the inverse of the matrix in the linear solve and perform matrix-vector multiplication online. All the numerical operations are computed using the embedded processor. For latency, see Figure 3 with label SW_invert.

For all considered scenarios the CPU clock frequency was set to 667 MHz, while the FPGA was clocked at 100 MHz. We did not parallelize any operations on the FPGA for this example to have a fair comparison with a pure software-based implementation on an embedded processor.

As illustrated in Figure 3, implementation on an FPGA with synthesis directives has the least latency, followed by a heterogeneous platform and a software implementation with LDL factorization. This shows the trade-off between a pure FPGA-based implementation and a pure software-based implementation. It is important to note here that an FPGA-based implementation without synthesis directives has a higher latency than an LDL-based pure software implementation. Since SPLIT is tailored to an algorithm, it performs synthesis directives during code generation and thus is ready to deploy for end users. It is also interesting to observe from Figure 3 that solving the linear system based on precomputing the inverse in a software-based implementation has the worst latency. We also note here that to make a library-free implementation, we apply a custom LDL factorization, which exploits sparsity. Since the linear system solve step has a sparse matrix, the factorization is sparse as well. Thus, the time taken for a factorization-based solver is better than computing the inverse and performing matrix-vector multiplication. However, due to lack of definite pattern in-fill, the forward backward solve is not suitable for implementing on FPGAs. A heterogenous implementation is a good trade-off between a pure FPGA implementation and implementation on embedded processor. This is due to the fact that the main computational bottleneck is performed on the FPGA.

We summarize resources used in Tables 4–7. It is worth noting that a synthesis directive-based implementation uses similar amounts of resources as one without a synthesis directive, while offering much better latency. A heterogeneous implementation uses less resources compared to a pure FPGA-based implementation, while suffering in latency.

Table 5. LUT usage in %

	Problem 1	Problem 2	Problem 3
PL_directive	19.70	22.42	25.72
PL	19.67	21.49	24.74
SOC	3.51	3.27	3.34

Table 6. FF usage in %

	Problem 1	Problem 2	Problem 3
PL_directive	7.54	8.86	11.76
PL	7.66	8.70	11.62
SOC	1.44	1.33	1.34

Table 7. BRAM usage in %

	Problem 1	Problem 2	Problem 3
PL_directive	2.68	9.82	50.53
PL	2.15	9.46	50.18
SOC	1.60	6.60	46.61

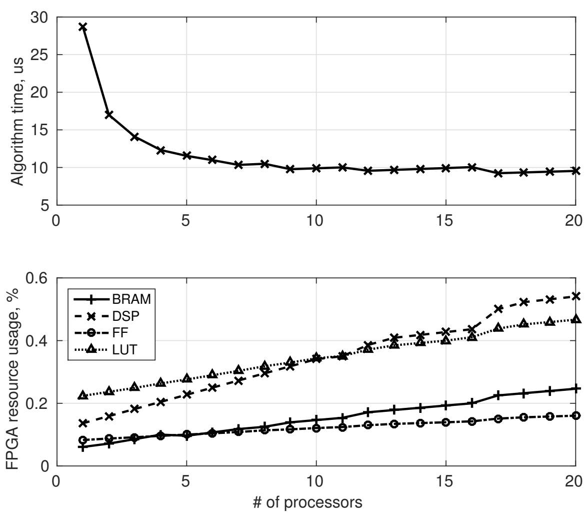

Fig. 4. Latency vs Resources trade-off

6.2 Trade-off : Latency versus Resources

In this example, we illustrate the trade-off between resource usage and latency. We randomly generate an MPC problem of form (3) with 4 states, 2 inputs and horizon length of 5 with box constraints on inputs. We solve the optimization problem using an accelerated version of the alternating minimization algorithm and implement on an FPGA. Varying the number of parallel processors for solving the linear system allows trading off FPGA logic usage against latency, as shown in Figure 4. As we increase parallelization, the latency improves (3x faster) at the cost of using more resources. Selecting the number of parallel processors for a particular application one has to keep in mind Ahmdal's law, which states that overall algorithm parallelization speedup is limited by sequential part of the algorithm. This explains why after reaching a certain point parallelizing computations does not improve performance.

7. CONCLUSIONS

This paper presented a code generation tool for software, hardware and heterogeneous implementations of predictive control algorithms using operator splitting methods. Experimental results confirmed that generating synthesizable hardware-tailored C code allows achieving a 3x to 11x speedup with hardware realizations compared to pure software implementations. Moreover, it was shown that splitting the workload between software and hardware allows one to achieve a compromise between latency and computational resource utilization.

Further work will be focused on developing systematic techniques for automating selection of design parameters, e.g. the degree of parallelism. This will allow avoiding situations illustrated as in Figure 4, where starting from a certain point, parallelizing computations does not improve performance due to Ahmdal's law.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' guide*, volume 9. SIAM.

Condat, L. (2013). A primal–dual splitting method for convex optimization involving Lipschitzian, proximable and linear composite terms. *Journal of Optimization Theory and Applications*, 158(2), 460–479. doi:10.1007/s10957-012-0245-9. URL `http://dx.doi.org/10.1007/s10957-012-0245-9`.

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

Davis, T.A. and Hu, Y. (2011). The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), 1:1–1:25. doi:10.1145/2049662.2049663. URL `http://doi.acm.org/10.1145/2049662.2049663`.

Gabay, D. and Mercier, B. (1976). A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 2(1), 17 – 40. doi: http://dx.doi.org/10.1016/0898-1221(76)90003-1.

Lawson, C.L., Hanson, R.J., Kincaid, D.R., and Krogh, F.T. (1979). Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3), 308–323. doi:10.1145/355841.355847. URL `http://doi.acm.org/10.1145/355841.355847`.

Stathopoulos, G., Shukla, H., Szucs, A., Pu, Y., and Jones, C.N. (2016). Operator splitting methods in control. *Foundations and Trends® in Systems and Control*, 3(3), 249–362. doi:10.1561/2600000008. URL `http://dx.doi.org/10.1561/2600000008`.

Suardi, A., Constantinides, G.A., and Kerrigan, E.C. (2015). Fast FPGA prototyping tool for embedded optimization. In *Proc. European Control Conference*.

Tseng, P. (1991). Applications of a splitting algorithm to decomposition in convex programming and variational inequalities. *SIAM Journal on Control and Optimization*, 29(1), 119–138. doi:10.1137/0329006.