A Many-Core based Execution Framework for IEC 61131-3

Matthias Becker*, Kristian Sandström†, Moris Behnam*, Thomas Nolte*†
*MRTC / Mälardalen University, Västerås, Sweden
{matthias.becker, moris.behnam, thomas.nolte}@mdh.se
†ABB Corporate Research, Västerås, Sweden
kristian.sandstrom@se.abb.com

***Abstract*—Programmable logic controllers are widely used for the control of automation systems. The standard IEC 61131-3 defines the execution model as well as the programming languages for such systems. Nowadays, actuators and sensors connect to the programmable logic controller via automation buses. While such buses, as well as the sensors and actuators, become more and more powerful, a shift away from the current distributed operation of automation systems, close to the field level, becomes possible. Instead, execution of complex control functions can be relocated to more powerful hardware, and technologies. This paper presents an execution framework for IEC 61131-3, based on a many-core processors. The presented execution model exploits the characteristics of the IEC 61131-3 applications as well as the characteristics of the many-core processor, yielding a predictable execution. We present the platform architecture and an algorithm to allocate a number of IEC 61131-3 conform applications. Experimental as well as simulation based evaluation is provided.**

I. Introduction

Automation systems are constantly gaining in complexity, while at the same time they must be able to quickly adapt to marked changes. This is especially important for smaller industries, where flexibility and adaptability is crucial in order to stay competitive.

Communication mechanisms are one of the backbones of automation systems. Traditionally, bus based systems dominated the automation industry. However, more powerful and flexible automation networks appear and allow the connection of thousands of actors and sensors to the same network, while still obtaining the required timing performance. An example of such a bus technology is the Ethernet based PROFINET [1], an overview of several other industrial Ethernet variants is provided in [2]. Those changes in the communication technologies open the possibility of computation further away from the field level, compared to how it is done in today's automation systems. On the other hand, many sensors and actuators have become *intelligent*, already today. This means, they are equipped with small microcontrollers, allowing them to do basic data processing inside the sensor, and they are able to connect directly to the new bus technologies.

Both trends, the increased flexibility of automation systems, and the increased capabilities of sensors and actuators paired with more capable bus systems, pave the way for cloud-based solutions in an industrial automation environment. Several solutions on different levels of an automation process are already proposed [3]. Having basic data processing done at the lowest level, e.g. directly at the sensor, and a connection to capable networks, allows the reallocation of applications, such as control loops, away from the field level into so called *compute pools*. Such compute pools are decentralized with enough compute power for a large number of applications, while providing the required flexibility to quickly adapt to changes of the applications requirements. This has several benefits. On the one side, the shop floor level gets more flexible. Changing control applications becomes merely a problem of reconfiguration in the compute pool. Costs will decrease as well, the Programmable Logic Controller (PLC) is migrated to the compute pool, i.e. the need for physical PLCs will be decreased. This is already visible in the trend of so called *Soft PLCs* [4], where PLC systems are executed as applications within a legacy OS. Maintenance costs will be decreased as well. However, moving computations away from the process must be done with caution. If the compute pool is situated off site, the existing infrastructure of the network provider is used to connect to the compute pool. Delays on such networks, and thus the delays imposed on the control loops, are in general unpredictable. One alternative is the use of on premise compute pools, where knowledge about the network and its performance can be used to bound these delays.

In order to pave the way for cloud based solutions which satisfy the requirements of the automation domain, current trends in hardware architecture are exploited. Having a steady increase in the number of cores, implemented on one processor, allows for new techniques in order to consolidate applications. Additionally, such many-core processors come with low energy consumption and a massive amount of computational power [5] compared to single core solutions.

As main contribution of this paper, we present an execution framework for IEC 61131-3. This framework is based on a many-core platform, allowing the consolidation of legacy IEC 61131-3 applications. Additionally we present an algorithm to map a set of IEC 61131-3 applications to such a platform.

The remainder of the paper is organized as follows. In Section II we present related work. Section III presents the background and system model. In Section IV, the proposed execution model and framework are introduced, followed by an algorithm to allocate multiple IEC 61131-3 applications on such a platform. Finally, we present the evaluation of the proposed framework in Section V, followed by concluding remarks in Section VI.

The work presented in this paper is supported by the Swedish Knowledge Foundation via the research project PREMISE.

II. Related Work

Several works address the IEC 61131-3 standard. Beremiz, as an academic project, offers full capabilities to design and compile IEC 61131-3 conform applications to several hardware platforms [6], [7]. As multi-core processors become more common, their usage in the context of industrial automation is addressed as well. Recent efforts focus on the parallel execution of function blocks of one network, in order to minimize the finishing time of the application [8], [9], [10], [11]. This allows for shorter scan cycle times and therefore better control. In contrast, [12] and the framework presented in this paper linearize applications where the focus lies on consolidation of multiple legacy applications on the same PLC. The objectives are therefore orthogonal.

On the hardware side, the increasing demand for computational power leads to an architecture shift. A slow trend away from multi-core processors towards many-core processors is visible. Many-core processors have a large number of simple cores implemented on one die. This allows for tighter analysis [5] compared to today's complex multi-core processors [13]. In [14] the authors propose an operating system designed for such many-core processors. In their approach the cores are exclusively reserved for one service. This is viable, because future processors are expected to accommodate thousands of cores. Kalray's MPPA processor for example is expected to implement up to 1024 compute cores with its next generation [15].

III. Background and System Model

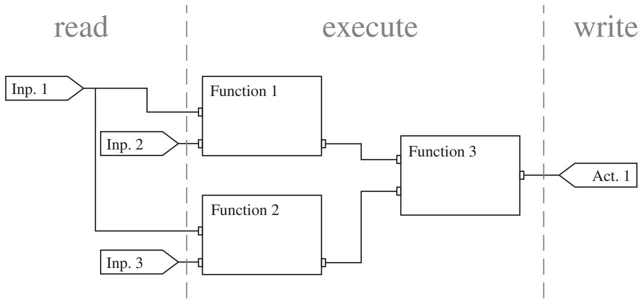
In this section we first discuss the basic concepts of automation applications following the IEC 61131-3 standard. Additionally we describe the hardware platform which will be used to base our execution framework on.

A. IEC 61131-3

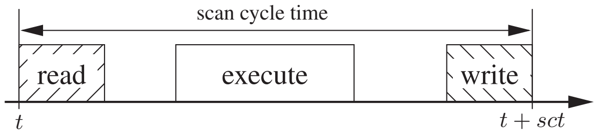
As an effort to provide one common basis to program industrial control systems, the International Electrotechnical Commission defined the standard IEC 61131-3 [16]. This standard defines four programming languages for the Programmable Logic Controller (PLC). In this paper, without loss of generality, we focus on the Function Block Diagram (FBD), as one of the four languages defined by the standard.

When using FBDs, different forms of program organization units can be used to define the program. *Functions* provide rudimentary functionality. They have no internal state and thus always yield the same output values for the same input values. *Function blocks* on the other hand are more complex. They can contain more elaborate functionality and have internal states. Both, functions and function blocks, have well defined interfaces and hidden internals. While this behavior is in line with the principles of component based software engineering other structural elements in IEC 61131-3 (global variables, direct addressed variables, access paths, etc.) introduce hidden interfaces, hindering a fully component based usage. Sünder identifies such hidden interfaces in [17].

Programs written using FBDs are networks of functions and function blocks, as shown in Fig. 1a. For execution, they are mapped to tasks of the operating system. Here, they are periodically or event triggered executed based on the *read-execute-write* semantic. Execution based on this paradigm is depicted in Fig. 1b. All input values are read to local memory at the beginning of the cycle, and all output values are written back at the end of the cycle. During the execution phase, only the local copies of the data are used. This execution model has several advantages. Data communication is only needed during the defined phases. Also, the jitter, experienced by the actuators, is reduced, allowing for a higher quality of the control loops.



(a) Example of a FBD network with separation in its three execution phases.



(b) Different execution phases of one application during one period.

Fig. 1: FBD network and the representative execution schedule.

The standard further defines a complete software model, where a *configuration* can be seen as highest level of abstraction, encapsulating all software needed to solve the designated task. A detailed definition of the software model and its individual parts can be found in the standard [16].

In this work, C represent one IEC 61131-3 configuration, and Θ represents one FBD network as depict in Fig. 1a. As one configuration usually consists of multiple FBD networks, C is represented by a set $\{\Theta_1, \Theta_2, \ldots, \Theta_n\}$, where n is the number of FBD networks. Each FBD network Θ is represented by the tuple $\Theta = \{\delta, T\}$. T represents the period, or scan cycle time, of the graph. δ is a directed graph $\delta = (\mathcal{F}, \mathcal{E})$. $\mathcal{F}$ represents the set of m functions or function blocks $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ composing the application. Each function f_i, has a weight w_{f_i} which is equal to its Worst Case Execution Time (WCET). The edges $\mathcal{E}$ represent the set of data dependencies in the network. One edge $e_{i,j} \in \mathcal{E}$ represents a data dependency between f_i and f_j, where f_i is writing and f_j is reading the data. An edge weight of w_{e_i} states the size of the data communicated each cycle. Each edge is further associated with a name, representing the data variables involved. Having the read-execute-write semantic, we add one node at the beginning of the graph and one node at the end of the graph, both of weight 0. All input variables which need to be read during the *read* phase are represented as edges from the input node to the respective nodes. All output variables of the application are represented by edges ending in the end node.

B. Hardware model

In this section we present the design of the compute node, used to execute the workload. The basic design can be

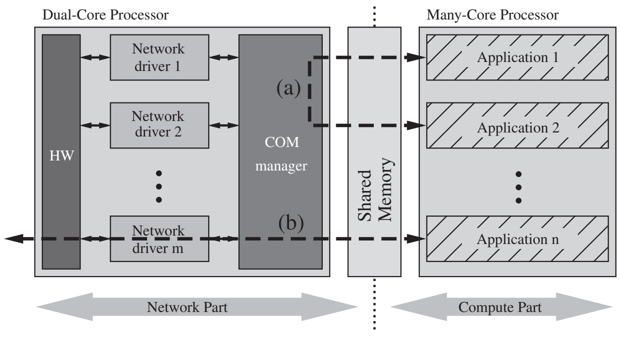


Fig. 2: Architecture view of one Compute Node.

seen in Fig. 2. It consist of two parts, this is in line with available architectures (see for example [18]). The network part, consisting of a powerful multi-core processor, is used to receive and transmit the sensor and actor data over the network. This part is also responsible to write and read data into the shared memory, where it can be accessed from the many-core based compute part. It thus can be seen as an abstraction layer between the data aggregation over the respective network and the execution of the IEC 61131-3 configurations. The compute part consists of a many-core processor, and it is responsible for the execution of the IEC 61131-3 configurations. It consists of a 2 dimensional mesh-based Network-on-Chip (NoC) of size $l \times l$ and a set of $l \times l$ compute cores $\mathcal{N} = \{N_1, N_2, \ldots, N_{l\times l}\}$. Each core, together with local memory, is connected to one of the NoC routers. Note, that the core local memory is usually magnitudes smaller than the off-chip memory but access to off-chip memory must nevertheless be handled with great caution since the induced delays are significant. We limit ourselves therefore to only use this core-local memory for code and data placement.

As shown in Fig. 2, the network part consist of multiple elements. As its main purpose is to abstract the data communication away from the compute part, it contains one or possibly multiple network drivers. A software layer, called communication manager, is used to handle the different communication forms. This communication can be platform local (a) or targeting an element reachable over one of the available networks (b). The communication manager is responsible to provide this abstraction. From an application point of view there is no difference between on-platform or platform to platform communication, since data always needs to be written into the shared memory from where it is picked up by the communication manager.

IV. Execution Model of the Compute Node, and Mapping of Function Block Networks to Cores

This section describes the execution model of the compute node. Precisely the execution of IEC 61131-3 configurations on the many-core based part of the node, followed by a heuristic algorithm to allocate applications to the cores.

A. *Dividing the platform in SCT-Cores*

Since our approach solely targets IEC 61131-3, we can exploit several characteristics of such applications. (i) One configuration consists of a large number of moderate sized FBD networks. (ii) The execution time of one function block is small and usually in the range of μs. (iii) While the Scan Cycle Time (SCT) of a network can be an arbitrary time, the number of different scan cycle times in real industrial applications is usually small (i.e., one SCT for fast control loops, normal control loops, and housekeeping functionality).

Based on those observations and the fact that we want to consolidate legacy applications, preserving their execution behavior rather than reducing their SCTs, we can conclude that parallel execution of function blocks of the same network is not beneficial for our approach. Thus, reducing the end-to-end delays of one application network is not subject of optimization. Having in mind the relatively small size of such networks, the gain in parallelization will diminish and at the same time produce greater complexity for the mapping of function blocks to cores, for the communication on the NoC, but also for the execution of said networks. Thus, application networks are mapped as a whole.

Since solely the IEC 61131-3 configurations are executed on the many-core platform all interrupts and overhead related to the network is handled by the network part of the node and thus on a different processor. Therefore, most demands on the operating system are on the network side and a small embedded operating system on each of the compute cores is sufficient. Employing fixed priority scheduling [19], as standard in industry, we have lower schedulability bounds than for example earliest deadline first scheduling [19]. However, exploiting the first and third observation and the given hardware characteristics, i.e. the large number of compute cores on the many-core platform, we can use cores to execute function blocks of one scan cycle time only. Having relatively many, but small, application networks, allows us to allocate these independent elements on any core of same scan cycle time.

From a scheduling point of view, this is beneficial, since it leads to simpler systems with possible utilization of up to 100%. The schedulability condition for a core i with scan cycle time SCT_i can be formulated by Inequation 1. A core can schedule all its FBD networks, as long as the time taken for reading and writing all global variables plus the execution of all function blocks is less or equal to the scan cycle time.

$$SCT_i \leq T_{Read} + T_{Execute} + T_{Write} \qquad (1)$$

B. *Heuristic algorithm to allocate FBD networks to compute cores*

Based on the execution model described in the previous section, we now propose an algorithm to distribute a set of IEC 61131-3 configurations $\Omega = \{C_1, C_2, \ldots, C_n\}$ to the cores of a compute node. The input for the mapping algorithm is a set of k compute cores and a set of FBD networks $\mathcal{A} = \{\Theta_1, \Theta_2, \ldots, \Theta_m\}$. $\mathcal{A}$ contains all FBD networks Θ of all configurations $C_i \in \Omega$. As discussed before, FBD networks are always executed on the same core. Thus, as a first step for the mapping algorithm, we transform a FBD network Θ into its serialized version θ. A serialized FBD network θ can then be described by the tuple $\{C, T, \mathcal{E}_{input}, \mathcal{E}_{output}\}$, where C represents the WCET of the serialized network and T represents its scan cycle time. Since exchange of parameters on the same core is fast compared to core-to-core communication, we neglect those overheads and we therefore only build the sets $\mathcal{E}_{input}$ and $\mathcal{E}_{output}$, in order to represent the global variables which are read or written during one scan cycle. C_i can be computed in a simple way: $C_i = \sum_{\forall f_k \in \mathcal{F}_i} w_{f_k}$.

As discussed before, one scan cycle of an application network consists of three parts (see Fig. 1b). Having only application networks of the same scan cycle time on one core allows us to group the respective parts of all application networks on the core. The utilization of a core $i \in [1,k]$ can then be computed by: $U_i = \frac{T_{Read,i}+T_{Execute,i}+T_{Write,i}}{SCT_i}$, where SCT_i is the scan cycle time of the core. $T_{Read,i}$ describes the time taken to read all variables from the shared memory. $T_{Execute,i}$ describes the total execution time of all functions and function blocks, and $T_{Write,i}$ represents the time taken to write all variables to the shared memory. Equation 2 shows the detailed calculations for the different parts.

$$U_i = \frac{\sum_{\forall e_k \in \mathcal{R}_i}(w_{e_k} \cdot \alpha) + \sum_{\forall \theta_i \in N_i} C_i + \sum_{\forall e_g \in \mathcal{W}_i}(w_{e_g} \cdot \beta)}{SCT_i} \tag{2}$$

$\mathcal{R}_i$ contains all global variables which are read by any of the FBD networks mapped to the core N_i. Similar, $\mathcal{W}_i$ contains all global variables which are written by any of the FBD networks mapped to the core N_i. Since the weight of the edges in $\mathcal{R}_i$ and $\mathcal{W}_i$ represent the size of the data, we multiply them by a parameter α and β respectively. Those factors represent the time it takes to read/write one data unit from/to the shared memory. Note that multi-core processors often implement different networks for read and write access and they commonly execute at different frequencies. Since, at this time, we do not know the traffic patterns of the cores connected to the NoC, we can not compute exact worst case delays of the messages used to access the shared memory. α and β are therefore used to give sufficient approximations.

Having serialized FBD networks reduces the complexity of the problem. The allocation algorithm now only needs to allocate the serialized *blocks* to the cores. Several requirements need to be considered. (i) The first FBD network mapped to a core dictates the scan cycle time of the core. I.e., all consecutive FBD networks mapped to that core must have the same scan cycle time. (ii) The mapping algorithm should use as few cores as possible. This will leave more space for FBD networks added afterwards. (iii) Additionally to the execution of function blocks on the cores, the cores need to read and write the input and output values once each scan cycle time. This time needs to be considered during the mapping as well. (iv) Memory requirements of the global variables required by the FBD networks, which are mapped to a core, need to be taken into account. The size of the shared memory is magnitudes larger and allows for allocating data equivalent to all local memory of the compute cores. Thus, while mapping, only the core local memory is taken into account.

Allocating tasks on many-core processors is a complex problem itself. It is possible to reduce the bin-packing problem to the problem at hand and thus, the problem is $\mathcal{NP}$-hard [20]. Heuristic algorithms are therefore used to find solutions for such problems. Having the second requirement, we base our algorithm on the First Fit Decreasing (FFD) algorithm, because it utilizes a low number of cores. Since we do not assign FBD networks of different scan cycle time to the same core, we can split the assignment algorithm in two parts. The first part, presented in Alg. 1, receives a set of FBD networks $\mathcal{A}$, which are of same scan cycle time, and assigns them to a set of cores $\mathcal{N}$. As third parameter the algorithm received $maxMem$ which states the maximal available memory space for global variables in the core local memory of each core. The algorithm first sorts the FBD networks by their WCET and saves the sorted list in $\mathcal{L}$ (line 4). Then, as long as there are FBD networks to assign, the algorithm checks the utilization, where U_{N_i} is the current utilization of the core in focus and U_{θ_1} is the utilization of the first FBD network in $\mathcal{L}$, and the available memory of all previously used cores. The utilization check is based on Equation 2 and the memory check is done by a function `memReq()` which returns `true` if the memory can be allocated. If a core can accommodate the additional utilization and memory requirement of the first, and thus largest FBD network, this network is assigned to that core and subsequently removed from the set. For the next FBD network all cores are checked again (line 18). Once all FBD networks are assigned to cores, the function returns the set of still empty cores (line 20). If an FBD network can not be assigned to one of the used cores, the algorithm adds a new core for this scan cycle time(line 13).

Algorithm 1 Assigning networks of the same scan cycle time

```
1: function = AssignSCT($\mathcal{A}$, $\mathcal{N}$, $maxM$)
2: $i = 1$;
3: $iMax = i$;
4: $\mathcal{L}$ =sortDescending($\mathcal{A}$, $C$);
5: while $\mathcal{L} \neq \emptyset$ do
6:     $\theta_1$ =getFirstElement($\mathcal{L}$)
7:     while $1 - U_{N_i} < U_{\theta_1} \vee$ !memReq($maxM$, $N_i$, $\theta_1$) do
8:         $i++$;
9:         if $i > |\mathcal{N}|$ then
10:            return $\emptyset$;
11:        end if
12:        if $i > iMax$ then
13:            $iMax = i$;
14:        end if
15:    end while
16:    assign($\theta_1$, $N_{ix}$);
17:    $\mathcal{L} = \mathcal{L} \setminus \theta_1$;
18:    $i = 1$;
19: end while
20: return $\{N_i | iMax < i \leq |\mathcal{N}|\}$;
21: end function
```

The main mapping algorithm is presented in Alg 2. It also receives a set of FBD networks $\mathcal{A}$, a set of cores $\mathcal{N}$, and the maximal memory available on each core $maxM$ as input parameters. As first step, the algorithm converts the FBD networks into their sequential representation and saves them in the set $\mathcal{L}$ (line 2). Then it assigns the FBD networks until they are all assigned. First, the algorithm checks if the set of cores is non-empty. If cores are available, the lowest scan cycle time in $\mathcal{L}$ is computed (line 7). Based on this scan cycle time, the subset of all FBD networks $\in \mathcal{L}$, of said scan cycle time, is saved in $\mathcal{K}$ (line 8). Those FBD networks are then removed from the set $\mathcal{L}$ (line 9). In line 10, we call the previously described algorithm to assign the set of FBD networks of same scan cycle time. Note, that this function returns the set of not used cores, these cores are then used for FBD networks of other scan cycle times.

Algorithm 2 Assigning networks to cores

```
1:  function = MapToCores($\mathcal{A}$, $\mathcal{N}$, $maxM$)
2:  $\mathcal{L}$=CreateSequentialRepresentation($\mathcal{A}$)
3:  while $\mathcal{L} \neq \emptyset$ do
4:      if $\mathcal{N} = \emptyset$ then
5:          return error
6:      end if
7:      $T_{low} = \{T | T \leq T_k \in \mathcal{L}\}$
8:      $\mathcal{K} = \{a_i | T_i = T_{low}, a_i \in \mathcal{L}\}$
9:      $\mathcal{L} = \mathcal{L} \setminus \mathcal{K}$
10:     $\mathcal{N}$ = AssignSCT($\mathcal{K}$, $\mathcal{N}$, $maxM$)
11: end while
12: return success
13: end function
```

V. Evaluation

The evaluation of the proposed framework was carried out partly based on real measurements and partly based on simulations. We first describe measurements to quantize the induced delays brought by the shared memory communication. Later we present different properties of the system, based on a case study.

A. Delays induced by the shared memory on the compute node

One critical part of the hardware architecture (see Section III-B) is clearly the shared memory. While the execution of IEC 61131-3 can be parallelized on the many-core processor, the data flow to and from the sensors, actuators, and other applications over the host processor is realized using a shared memory. Measurements on a hardware platform [18] similar to the one assumed in this work were conducted in order to verify the applicability of such a design.

The hardware platform consist of a dual core ARM Cortex A9 CPU implemented on the Zynq-Z7010 processor [21], acting as host and running a Linux based operating system. A 16 core version of Adapteva's Epiphany processor [22] constitutes the compute part of the platform. Communication between the ARM and the Epiphany processor is done using a shared memory. We conducted measurements with two different data sizes, 512 kB and 1024 kB, and with a varying number of active slave cores. For each data point 1000 measurements where taken.

The time taken for the *read* and *write* sequence of a scan-cycle was measured. One measurement consists of the network part writing the data into the shared memory. Once the data is in the shared memory, a varying number of cores on the epiphany processor is notified about the arrival of the data in the shared memory. After this trigger each compute core loads the data to the core-local memory. Since we are just interested in the read and write times, those cores immediately write the data back into the shared buffer, followed by notifying the dual core processor. After the notification was received from all active compute cores, the network processor reads the data from the shared memory.

The delay encountered during those operations is shown in Fig. 3. The x-axis depicts the number of compute cores used for each measurement. It can be seen that the delay encountered for 1024 kB of data increases slightly faster than that for the 512 kB of data. The reason is the increased interference on the NoC, additionally to the increase in data.

TABLE I: Parameters for the generation of FBD networks

Parameter	Min	Max
Execution time	0.122 ms	1.223 ms
Input Parameter	1 byte	50 byte
Output Parameter	1 byte	50 byte

B. Case study

In this section we present different properties of the system. We generate a number of IEC 61131-3 configurations based on parameters commonly found in industrial automation systems. Those configurations are then mapped to the hardware platform.

Table I presents the parameters for the generation of the FBD networks. The values for execution times are based on the measurements presented in [23]. An average function block execution time of 12.24 µs was chosen based on their measurements, and the values for the execution times were selected accordingly, assuming between 10 and 100 function blocks in one FBD network. The minimal and maximal size of input and output parameters is shown in the table as well. The value for each FBD network was chosen in those ranges based on a uniform distribution. We further assume that FBD networks operate on a small set of scan cycle times, $\{10\,\text{ms}, 20\,\text{ms}, 100\,\text{ms}, 200\,\text{ms}\}$ with uniform distribution.

The platform parameters were chosen based on the Epiphany processor [22]. The compute part implements 16 cores. Each core has 16 kB of local memory which is used for both, code and data. For the experiments we allow the usage of 4 kB for input and output data per compute core. We further select the parameters α and β of Equation 2 based on the throughput measurements. For the experiments we assume the same operation frequency on read and write network, which results in $\alpha = \beta = 1.145 \frac{\mu s}{\text{byte}}$.

Because the mapping algorithm does not distinguish between FBD networks of different configurations we only generate one configuration, consisting of 800 FBD networks. Each of these FBD networks are generated based on the values above. The generated configuration is then assigned to the compute cores. The algorithm assigns FBD networks of SCT 10 ms to 6 cores, SCT 20 ms to 4 cores, and SCT 100 ms and

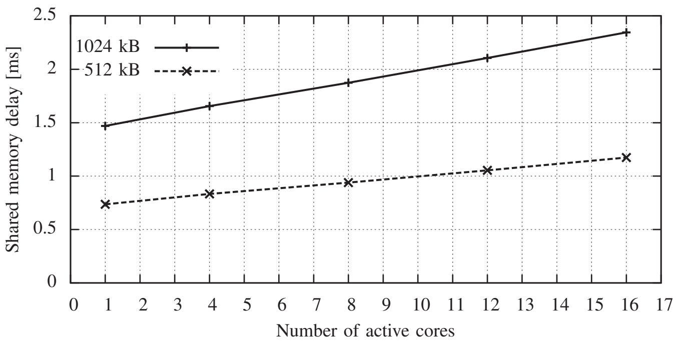
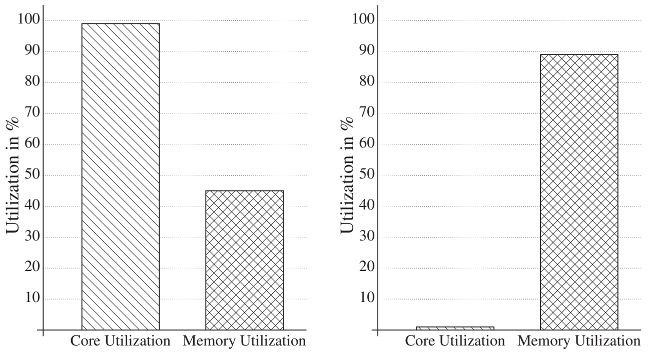


Fig. 3: Measurement of the communication delay between network and compute part for different data sizes, read and written by a varying number of compute cores.



(a) Scan cycle time of 10 ms. (b) Scan cycle time of 200 ms.

Fig. 4: Comparison of the average core and memory utilization of the cores belonging to one scan cycle time.

200 ms are each assigned to 3 cores.

In this experiment, the two limiting factors on one compute core are examined. Namely the core utilization and the utilization of the memory region used to save the input and output parameters. Fig. 4 depicts the two corner cases, namely the average distributions on all cores operating at scan cycle time 10 ms and 200 ms. We can see that the cores operating at 10 ms are almost utilized up to 100% while still more than half of the available memory is unused. On the other hand the cores operating at 200 ms have very low core utilization while the available memory is almost completely used. This observation is not surprising, since the execution time of the application is not dependent on the scan cycle time. Thus, a core executing FBD networks of a large scan cycle time can host significantly more FBD networks than a core executing at a low scan cycle time. So the relatively low size of core local memory will become the bottleneck. Note that this heavily depends on the used many-core processor. While the core local memory on the reference platform is small,other many-core processors provide much more core local memory. Kalray's MPPA-256 processor for example provides 128 kB of core local memory [5].

VI. Conclusion and Future Work

In this work we presented a many-core based execution framework for IEC 61131-3. The co-processor based architecture of most of today's available many-core platforms is exploited to divide the encountered work on a PLC. Network communication is carried out on the host processor and execution of the IEC 61131-3 code is done on the many-core co-processor. Executing only FBD networks of the same scan cycle time on one core reduces the scheduling complexity immense, this is possible due to the large number of cores on such many-core processors.

Future work will focus on a prototype implementation as well as on the challenges of resource management and load balancing in a compute pool of multiple such nodes. We think powerful hardware architectures, such as many-core based systems, will allow a seamless transition to cloud-based solutions. The architecture characteristics of such platforms allow for simpler execution models, yielding less overhead and more predictability.

References

[1] *PROFINET*, last access April 2015, available at http://www.profibus.com/technology/profinet/.

[2] J.-D. Decotignie, "The many faces of industrial ethernet [past and present]," *IEEE Industrial Electronics Magazine*, vol. 3, no. 1, pp. 8–19, 2009.

[3] O. Givehchi, H. Trsek, and J. Jasperneite, "Cloud computing for industrial automation systems - a comprehensive overview," in *IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, 2013, pp. 1–4.

[4] O. Givehchi, J. Imtiaz, H. Trsek, and J. Jasperneite, "Control-as-a-service from the cloud: A case study for using virtualized PLCs," in *10th IEEE Workshop on Factory Communication Systems (WFCS)*, 2014, pp. 1–4.

[5] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Conference on Design, Automation & Test in Europe (DATE)*, 2014, pp. 97:1–97:6.

[6] E. Tisserant, L. Bessard, and M. de Sousa, "An open source IEC 61131-3 integrated development environment," in *5th IEEE International Conference on Industrial Informatics (INDIN)*, 2007, pp. 183–187.

[7] *Beremiz*, last access April 2015, available at http://www.beremiz.org/.

[8] A. Canedo, L. Dalloro, and H. Ludwig, "Pipelining for cyclic control systems," in *16th International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2013, pp. 223–232.

[9] A. Canedo, H. Ludwig, and M. Al Faruque, "High communication throughput and low scan cycle time with multi/many-core programmable logic controllers," *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 21–24, 2014.

[10] A. Canedo and M. Al-Faruque, "Towards parallel execution of IEC 61131 industrial cyber-physical systems applications," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 554–557.

[11] A. Monot, A. Vulgarakis, and M. Behnam, "Pasa: Framework for partitioning and scheduling automation applications on multicore controllers," in *IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–8.

[12] A. Vulgarakis, R. Shooja, A. Monot, J. Carlson, and M. Behnam, "Task synthesis for control applications on multicore platforms," in *11th International Conference on Information Technology: New Generations (ITNG)*, 2014, pp. 229–234.

[13] G. Fernandez, J. Abella, E. Quiñones, C. Rochage, T. Vardanega, and F. J. Cazola, "Contention in multicore hardware shared resources: Understanding of the state of the art." in *14th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2014.

[14] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[15] *Kalray MPPA Many-Core*, last access April 2015, available at http://www.kalray.eu/IMG/pdf/FLYER_MPPA_MANYCORE.pdf.

[16] *IEC 61131-3*, International Electrotechnical Commission Std., 2003.

[17] C. Sunder, A. Zoitl, J. Christensen, H. Steininger, and J. Rritsche, "Considering iec 61131-3 and iec 61499 in the context of component frameworks," in *6th IEEE International Conference on Industrial Informatics (INDIN)*, 2008, pp. 277–282.

[18] *Parallella Board*, last access April 2015, available at www.parallella.org.

[19] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[21] *Zynq-7000 Silicon Devices*, last access April 2015, available at http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices.html.

[22] *Epiphany Architecture Reference*, Adapteva Inc., Adapteva Inc. 1666 Massachusetts Ave, Suite 14 Lexington, MA 02420 USA, 2013.

[23] M. Oriol, M. Wahler, R. Steiger, S. Stoeter, E. Vardar, H. Koziolek, and A. Kumar, "FASA: A scalable software framework for distributed control systems," in *3rd International ACM SIGSOFT Symposium on Architecting Critical Systems (ISARCS)*, 2012, pp. 51–60.