BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 64, No. 1, 2016
DOI: 10.1515/bpasts-2016-0018

On the systematic method of conditional control program execution by a PLC

A. MILIK*, M. CHMIEL, and E. HRYNKIEWICZ

Institute of Electronics, Silesian University of Technology of Gliwice, 16 Akademicka St., 44-100 Gliwice, Poland

Abstract. The paper presents an original idea of the selective control program execution that allows significant response time reduction. The exhaustive analysis of the PLC program performance is given. An analytic approach explains the idea of the selective control program evaluation and gives the requirements for its feasibility. There is presented a systematic and formal method of program analysis based on a data flow graph approach. The method generates acyclic graph from the control program that is subject of optimization, variable allocation and instruction generation. The graph approach allows determining variables dependencies and task partitioning required by selective program execution. The method utilize the hardware supported variable changes detection. It is transparent for system operation and enables evaluation of blocks that require update.

Key words: control program, PLC, LD, IL, FPGA, compiler, control program optimization, PLC programming, flow graph.

1. Introduction

A PLC is a custom computer for implementing a control programs. It is able to handle multiple independent processes. In opposite to typical (imperative) programming that focuses on sequential processing of a particular task a PLC handles multiple control tasks [1]. The programming concept enables distributing computing power uniformly among all tasks. Each task is constructed in a form of function that determines the control value in discrete approach.

The control system is described by three sets of variables X, Y, Q. Items of the set X are associated with control input signals, while items of the set Y are associated with output signals. Variables of the set Q store an internal state of controlled process. The system is described by functions f and g that determines variables value of Y and Q sets:

$$\begin{aligned} Q_{n+1} &= f(X, Q_n), \\ Y &= g(X, Q_n). \end{aligned} \quad (1)$$

The discrete operation of the system is emphasized by variables value of the set Q. Indexes n and $n+1$ denotes current and next cycle value respectively.

The PLC is a part of control system (Fig. 1) that together with sensors, actuators and controlled object constitutes a control system. The response time of the controller should be minimized in order to immediately react for the changes of the input conditions [2, 3]. This requirements differs from a typical computing application working without direct feedback. A calculation throughput is a primary goal in such system while a calculation latency is less important. This approach favours pipelined constructions that distribute calculations into several processing steps. Samples are processed in overlapped fashion with possible the highest throughput and stages utilization introducing processing latency between input and output samples. A PLC operates in a closed loop with controlled object and must not operate in pipelined fashion. A tight correspondence between control algorithm and object response is required [4–6].

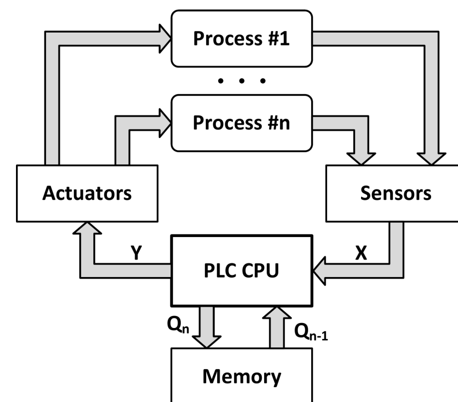


Fig. 1. A PLC based control system

The PLC computation cycle is shown schematically in the Fig. 2 [7]. It is equipped with proprietary operating system that manages entire system operation. The block named Program Execution is responsible for executing control algorithm delivered by user. The cycle dedicated for collecting input signals transfer inputs value to process image memory protecting against data race. The calculation results are dispatched from process image memory at the end of processing eliminating accidental output switching during processing. Remaining part of the program loop is responsible for performing system and housekeeping functions. For further performance considerations these tasks has been grouped into system functions.

*e-mail: adam.milik@polsl.pl

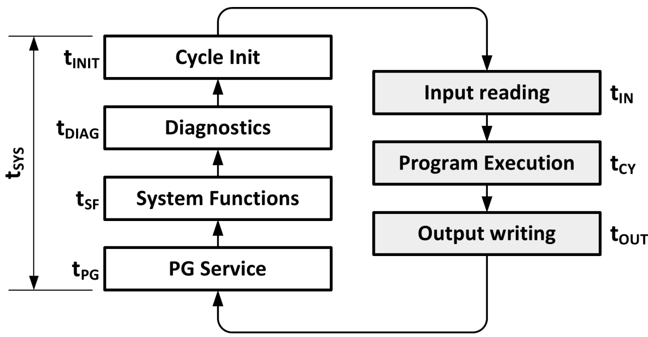


Fig. 2. Typical computation cycle of a PLC

This paper focuses on reducing user program execution time. The chapter two evaluates the program execution model. It is aimed at determining conditions of the control program recalculation. The efficiency of event driven calculations is shown. The chapter three brings the formal methodology of control program analysis, optimization and dependence tracing. It utilize the original graph representation suitable for exhaustive control program evaluation. The problem of program partitioning is also solved with graph based approach. The section four discuss a specific hardware support that significantly improves the event driven system and reduces programmatic overhead with determining program blocks to be recalculated. Finally the paper is concluded with an example that illustrates the method application and the obtained result.

2. The selective control program processing

2.1. The PLC performance. For comparison purposes of the PLC efficiency manufactures describe its performance by a scan time that is an equivalent of execution of 1000 instructions. The test set is a mixture consisting of 70% of logic instructions and 30% of other. It is based on analysis of a representative programs set given in [8]. This factor does not describe the controller throughput for a particular control system. The PLC program performance and the PLC system itself is measured by response time for an input signal change. The diagram (Fig. 3) shows graphically execution cycle. The interval marked t_{EV} is a range of time for the event (a variable value change) to be processed in the following cycle. The response time of the PLC vary and depends on the moment of time an event occurs. It assumes that inputs state are latched and collected at the beginning of a cycle (t_{IN}) and results are dispatched at the end (t_{OUT}). All system activities given in detail are aggregated into a single block of system activities (t_{SYS}). Presented model allows for analytic approach to the response time. The response time for an input event falls into a range determined by the shortest ($t_{R\min}$) and the longest time ($t_{R\max}$):

$$t_{R\min} = t_{IN} + t_{CY} + t_{OUT} = t_{SCAN} - t_{SYS},$$
$$t_{R\max} = t_{SYS} + 2\left(t_{IN} + t_{CY} + t_{OUT}\right) = 2t_{SCAN} - t_{SYS}. \tag{2}$$

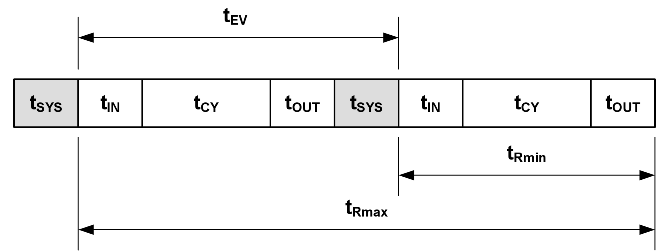


Fig. 3. The PLC response and observation times definitions in reference to computation cycle

For the safety reasons the longest one ($t_{R\max}$) is taken into considerations. The worst case analysis demonstrates that the response time of a PLC is almost equal twice the scan time (t_{SCAN}).

It could be noticed that the response time is radically reduced to the most optimistic case when an event occurs close to input reading cycle. The optimistic response time is even shorter then single scan time. For a periodically changing signal its period must be greater than a scan time. From the other hand the control process described by (1) evaluates sets Q and X. Variables belonging to the set Q store an internal state of a controlled process. Unpredictable execution time of a control program disables use of Q items for creating time dependencies [9]. The time dependencies are implemented with use of dedicated hardware timer units. In presented approach timers outputs are associated with variables that belong to the set X even though they are internal components of a PLC. Finally, the control equation for two discrete moments of time when variables x_i are constrained to constant values is:

$$\underset{x \in X}{\forall} x_i = \text{const}: \begin{cases} f(X, Q_n) = f(X, Q_{n-1}) = Q = \text{const} \\ Y = g(X, Q) = \text{const} \end{cases} \tag{3}$$

As it is illustrated the control program calculation should be triggered only if there are changes detected in the variables value of set X. The set is associated with all types of input signals controlling the process (object sensors, timers).

2.2. Implementation outline of an event driven concept. Based on presented consideration the following question can be formulated. "*Is it possible to reduce response time introducing non-standard program execution approach ?*". This approach is presented and called an event driven processing. It was inspired by analysis of the process control given in [10]. The computation time of the program requires nonstandard implementation of emergency control due to very long execution time of the main loop.

The concept of an event driven processing is shown in Fig. 4. The idea assumes a conditional execution of program blocks provided a change in respective input set is detected. This approach influences many aspects of a PLC architecture. It can be introduced as early investigation of the idea that is implemented in a PLC program only. This enables to confirm its feasibility and estimate the performance. A custom compiler has been proposed that automatically implements the event driven processing based on an automatic user pro-

gram partitioning. This approach is further extended not only to the program construction but also to a design of a specific hardware architecture that supports the developed processing methodology. The hardware-software co-design approach further improves the performance by data transfer connected with changes detection. Precisely tailored hardware platform and specific compiler support allow to achieve the best performance with a described approach.

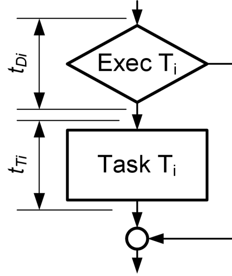


Fig. 4. Conditional execution of a task

2.3. The event driven method performance assessment. The term event driven processing requires formal justification of its feasibility and performance assessment. The most important question concerns the correctness of calculations performed by proposed model. The other objective is determination of the system performance in comparison to standard approach. All PLCs are ancestors of the micrprogrammable system that evaluates single argument at a time. The fundamental assumption requires that the shortest signal change time T_{CHG} is longer than the processing time expressed by t_{SCAN}

$$t_{SCAN} < T_{CHG}. \tag{4}$$

This assumption (and requirement) assures that controller is able to work out the response for all input signal changes and none of the events is missed. When the number of serviced objects and input signals is growing both the scan time and the response time are increasing. The increase of the response time reduces the application area of the controller determined by its throughput. For the single task (Fig. 5A) the response time is given as follows:

$$\begin{aligned} t_{RE\,\min} &= t_{IN} + t_E + t_T + t_{OUT}, \\ t_{RE\,\max} &= t_{SYS} + 2\left(t_{IN} + t_E + t_{OUT}\right) + t_T. \end{aligned} \tag{5}$$

When compared to a standard approach the difference execution time can be calculated. Let assume that the task execution time t_T remains unchanged. This allows to compare two approaches and determine requirements for event driven concept to be competitive to standard approach. Let's consider the response time differences:

$$\begin{aligned} \Delta t_{R\,\min} &= t_{R\,\min} - t_{RE\,\min} = -t_E, \\ \Delta t_{R\,\max} &= t_{R\,\max} - t_{RE\,\max} = t_T - 2t_E, \\ \Delta t_{Rav} &= t_T - 3t_E. \end{aligned} \tag{6}$$

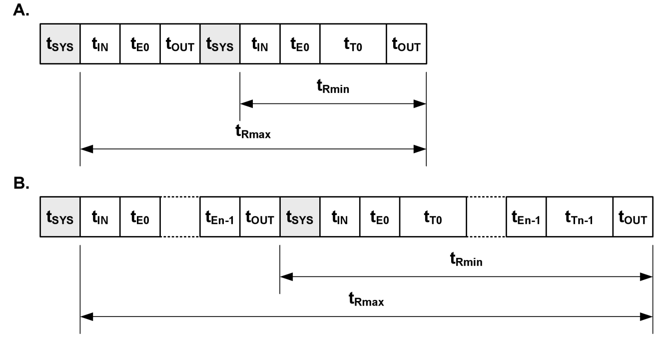


Fig. 5. Response of programmatic event driven processing

The response time increase (Δt) comparison allows to determine the limitation put on the conditional execution block. The minimum response time is increased by conditional execution test (negative decrease). The event driven programmatic approach due to its nature always reveals increase of minimal response time. The maximal response time allows to determine ratio between execution time of conditional test in reference to task execution time. The reduction of maximal response time will be observed for $t_E < \frac{t_T}{2}$. The average response time will be reduced when the conditional test execution time takes less than third part of execution time.

The single block control program was used for evaluation purposes. The control program is constructed from several tasks (blocks) that are executed in a given sequence by operating system [5, 6]. This observation is very important. There can be derived requirements for the event driven concept based on this observation. The task block is preceded with the conditional block that determines necessity of recalculating the task. The (3) gives a criterion for the task recalculation that is shown in form of a flow graph (Fig. 4). The figure depicts the execution time of the task triggering t_E and task execution t_T. The maximal response time for multiple tasks (Fig. 5B) when all require recalculation is given as follows:

$$\begin{aligned} t_{RM\,\min} &= t_{IN} + \sum_i \left(t_{Ei} + t_{Ti}\right) + t_{OUT}, \\ t_{RM\,\max} &= t_{SYS} + \left(t_{IN} + \sum_i t_{Ei} + t_{OUT}\right) + \sum_i t_{Ti}. \end{aligned} \tag{7}$$

When probabilistic approach is used than from the (4) the minimal signal period can be determined. We can also assume that not all signals are switching with the maximal frequency. This assumption will lead to the selective control task execution for particular scan. For simplicity let assume the uniform distribution of tasks execution time (all tasks require same period of time for computation) and only one task will be triggered during a single loop run. Let limit consideration to the program execution time t_P:

$$t_P = \sum_{i=0}^{n-1} t_{Ei} + t_{Tj}. \tag{8}$$

Let the maximal execution time of all conditional entry blocks is given as a fractional part of all computation execution. The factor d according to consideration is given as:

$$\sum_{i=0}^{n-1} t_{Ei} = \frac{1}{d}\sum_{i=0}^{n-1} t_{Ti}. \tag{9}$$

Than the program execution time is described in terms of task execution time and respective factors:

$$t_P = \frac{n+d}{nd}\sum_{i=0}^{n-1} t_{Ti}. \tag{10}$$

The relative program execution speedup η_P according to considered model is described as follows:

$$\eta_P = \frac{nd}{n+d}. \tag{11}$$

The above statement proofs the essence of program splitting into tasks. There are shown benefits coming from program splitting that is proportional to the number of tasks n. The event driven processing introduces additional cost of task identification. The efficiency of the method is proportional to the identification ratio given by d. In order to make the event driven processing competitive the task identification time must be minimized. This is addressed by a program generation methodology and a hardware assisted change detection.

3. Implementation of selective program processing

For the purpose of selective processing each program should consist of two blocks respectively linked that are a conditional trigger block and a standard processing block. This procedure can be applied manually to the existing program. A manual preparation is inefficient, complicated and time consuming. Moreover, the improperly constructed trigger block will not be able to call processing block when required, resulting in an inadequate behaviour and incorrect control processing.

3.1. Calculation dependencies in LD programs. The formula (3) describes a general requirement for event driven processing. In a PLC there are three sets of variables that are associated with inputs, outputs and internal markers. Let's consider an exemplary ladder schematics shown in Fig. 6. The inter rung dependencies are marked with dashed lines. The case A refers to recently evaluated rung scan. The waveform next to the diagram shows the event propagation from the input through all rungs. The case B shows the concept of pulse generation where the first rung refers to the variable calculated by the second rung. When the first rung is evaluated, the value of the $q4$ variable comes from previous scan cycle (not shown $t_n - 1$). The input $i4$ change is evaluated by both rungs and the value of $q3$ and $q4$ are assigned. Changes of $q4$ variable cause necessity of re-evaluating the network. When calculations are only triggered by variable associated with input signals there is a possibility of suppressing propagation of some events inside the program. The method for determining variable dependencies is an essential requirement for successful implementation of event driven processing. Authors have developed the method that not only allows to determine the variable dependencies but also allows to optimize and track dependencies in the entire control program.

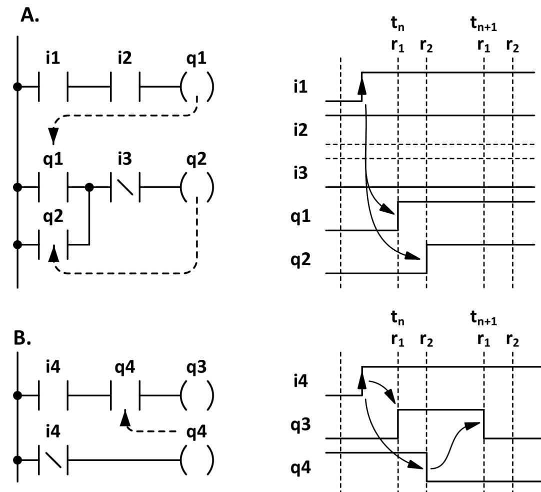


Fig. 6. Calculation dependencies in Ladder Diagrams

One of the possible approaches for a LD dependencies tracking is shown in work [11]. It attempts to investigate the sequential properties of the analysed network and translate it to the GRAFCET (SFC) representation. For this purpose dependencies and simultaneity graphs have been introduced. Those graphs have also been employed for a hardware implementation of LD [12]. Similar idea has been described by [13] for partitioning of the LD for multiple context hardware execution. The exhaustive program analysis demonstrates approach given in [14]. The specific derivative of a data flow graph has been developed and named the Enhanced Data Flow Graph – EDFG. It is applied for automatic analysis of PLC programs. The concept has been extended from LD to IL and SFC. The last method utilizing an EDFG is used for: extracting control program properties for event driven concept, program optimization and code generation.

3.2. Extracting LD program semantics with EDFG. In order to extract properties of the program an intermediate representation has been developed that allows to formulate algorithmic approach to LD program analysis and extracting its properties. It enables recording of logic and arithmetic operations. It tracks variables and operations dependences. The EDFG reveals parallel operations, branches and whole tasks. Depending on the target platform abilities and mapping methodology advantages can be taken from reviled parallelisms and task partitioning. Elimination of the dead code is an essential and natural effect of the control program graph representation. The described method enables optimized code generation that substantially differs from method presented in [15].

The EDFG is a directed acyclic graph. It is given by touple: $G = \langle V, E\rangle$ where: V is a set of nodes and E is a set

of directed edges. The directed edge e is described by an ordered triple $e = \langle v_{SRC}, v_{DST}, a \rangle$ where: v_{SRC} is a predeceasing node and v_{DST} is a successor node of the directed edge. The a is an attribute of the edge chosen from the set A. The set A consist of unary operations applicable for allowed nodes set. Attributed edges combines an assignment operation with a logic or arithmetic complement. This modification significantly simplifies logic and arithmetic transformations and optimizations.

The LD is converted into EDFG with application of mapping rules that are shortly given in the Fig. 7. The graph represents the sequential dependencies of calculation in entire program. Those dependencies and properties are difficult to track when direct code generation methods are applied. The first two cases show a variable value access. It distinguishes between unassigned (Fig. 7A) and assigned (Fig. 7B) variable access. There should be made a remark that an EDFG implements nodes responsible for value read and assignment separating them from operations.

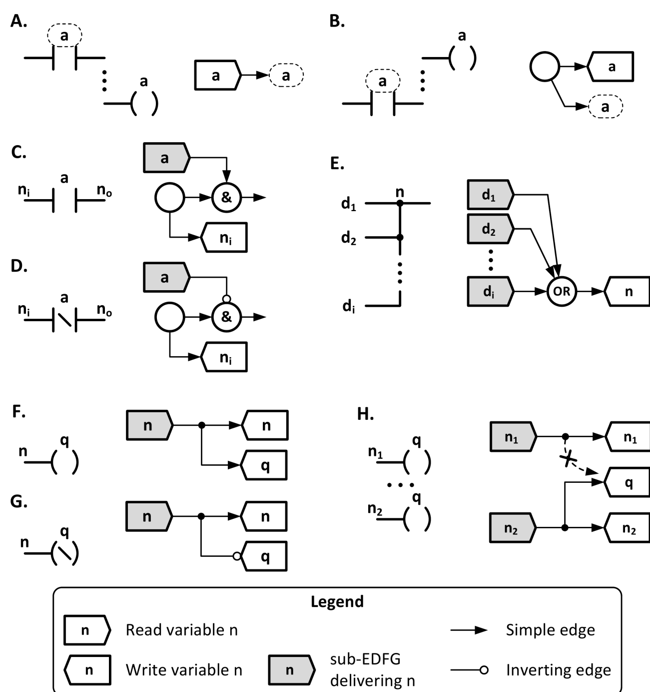


Fig. 7. LD to EDFG mapping

Value read nodes and value write nodes allow to link an EDFG with variables sets. The value access distinguishes between situations where access is made before assignment where the value comes from previous cycle or is worked out in current cycle as a result of preceding assignment. In the second case a value source node is accessed (Fig. 7B). If later the variable is reassigned the constructed graph tracks the specified calculation dependencies and current variable value source. Switches and nodes are the basic components of a network. The EDFG equivalent structures are shown in points C, D and E of the Fig. 7. Points C and D refer to the switch implementation. Attributed edges simplify handling of logic inversion (D). The schematic node is constructed with use of a logic OR operation. The node translation process is given in the diagram E of Fig. 7. Independently of the number of schematic node drivers the logic OR node is created that merges flow from multiple sources. Finally an assignment operation equivalent EDFG is shown. A write variable node is connected to driving node assigned recently. The EDFG is subject of optimization process where logic and algebraic rules are applied. The detailed consideration to EDFG mapping can be found in [14, 16]. The EDFG generation process and basic transformations are illustrated with following example.

An exemplary LD diagram is given in the Fig. 8A. The initial transformation perform substitution of elementary components with an equivalent EDFG structures. The process is continued until all components of input LD are substituted with EDFG nodes. The raw diagram just after generation process is shown in point B. Obtained EDFG is subject of optimization process based on node merge and constant propagation. The final result is shown in point C.

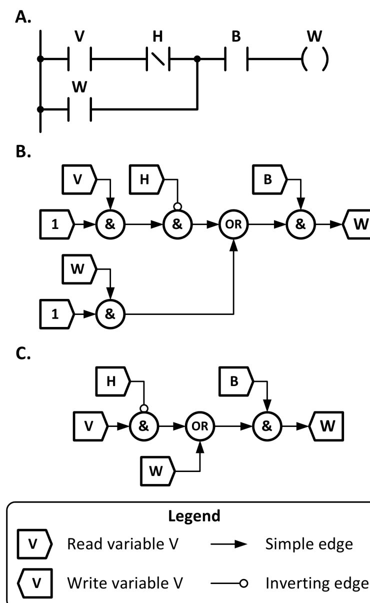


Fig. 8. The LD to EDFG transformation process and later optimization

3.3. The PLC code generation from the EDFG representation. The EDFG perfectly records the operations sequences given with a LD. The simplicity of creating and handling promotes using it instead of other partial analysis methods that not always guaranty achieving desired goals. The code generation process converts the EDFG representation into instructions sequence acceptable by a PLC CPU. The code generation process visits in reverse direction of edges all nodes starting from assignment nodes. This process is applied for variables associated with output and internal signals that were not optimized. The EDFG representation contains value read and write nodes associated with variables. A variable associated with internal marker is retained only if it is both assigned and referred to. A variable belonging to output set is retained when an assignment to it is made. A variable associated with

an input signal without reference (read access) is excluded (eliminated) from an input set.

The iterative code generation algorithm in simplified form for operation nodes has been depicted in Fig. 9. In the part A is given general case of possible node arguments. The node marked with grey colour is one under code generation process. The part B of the Fig. 9 shows diagrams similar to these used for the syntax definition [17]. In the rectangles are placed names of procedures that are called when particular path in code generation process is chosen. The rounded box contains a terminal part for code generation process in form of IL operators, arguments or parenthesis.

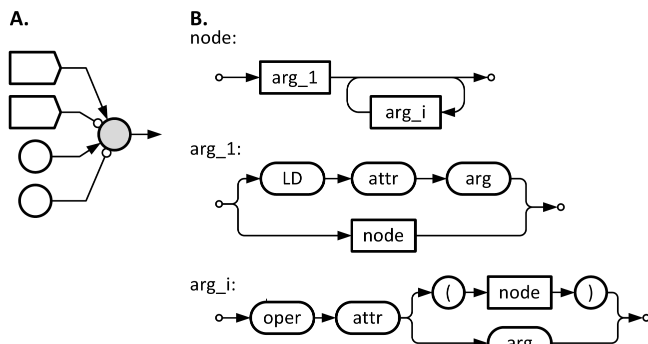


Fig. 9. A PLC code generation from the EDFG

The recursive process visits all argument nodes of currently evaluated node and follows respective procedure for each argument. The processing differs for the first argument, next arguments and already visited nodes. The first argument (*arg_1*) sets the initial value while remaining are processed in accumulative fashion according to IL language requirements. If the selected argument is an operation node the recursive approach is used. It calls the *node* procedure for it. The attribute (*attr*) reflects to logic inversion or arithmetic complement operation. For example the inversion attribute for an argument of a logic AND operation results in ANDN operation generation.

The nodes that are referred to multiple successors are calculated only once. A temporary variable is used for result sharing with all child nodes. This approach allows to take benefits from extracting common subexpressions. Formally the temporary variables are created for nodes that adjacency matrix contains more than one nonzero item in a row. In practical programmatic implementation the reference counter is used (instead of highly impractical adjacency matrix) [18].

The multiple times referred operation node requires creating temporary variable. The problem of temporary variables handling is shown in the Fig. 10. The variable lifetime associated with particular node extends from the calculation (a grey filled circle) to the last reference to it. The calculation model of the IL assumes that result is stored in accumulator register. A following operation does not require an internal variable creation. The result must be protected when multiple operations refers the same argument. The internal variables allocation process is based on the left edge algorithm [19]. Even though that algorithm is dedicated to routing an ASIC cells it can be successfully applied for variable management that is similar to a signal routing. Internal variables are created for all nodes that are referred more than once. In order to determine a variable lifetime the reference counter is used. It is initialized with a number of referring nodes. Whenever reference to the node is made the reference counter is decremented. If the reference counter reaches zero the internal variable is released (and can be reused). The internal variables set is initially empty. It is divided into subsets of free and assigned variables. New variable is created only if the free variable subset is empty. After completing the code generation process the variable set cardinality determines the number of internal variables. It corresponds to memory requirements imposed to the PLC. The number of required intermediate variables can be reduced by operation scheduling.

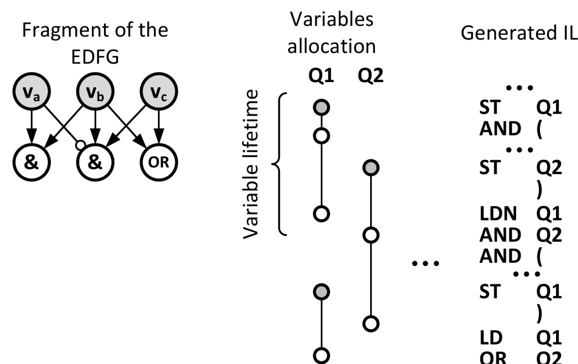
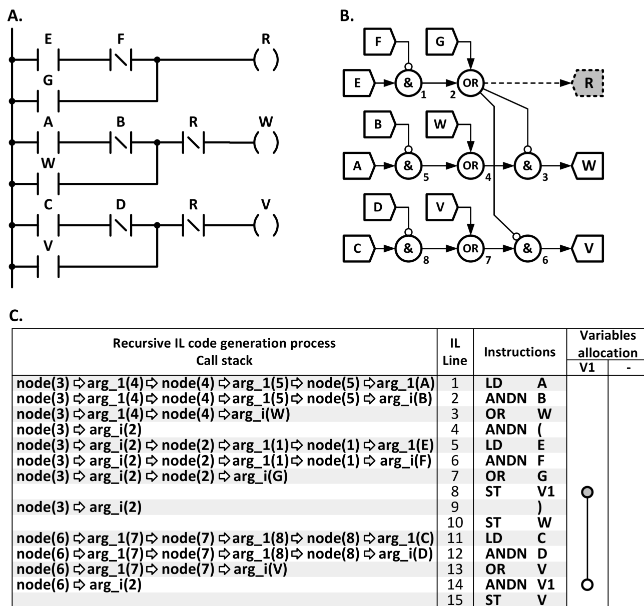


Fig. 10. The internal variables handling

An exemplary PLC code generation process with variable management is shown in the Fig. 11. The signal R is an internal signal. The LD program (A) is converted to the equivalent EDFG that is a subject of described instruction generation process for a PLC. The optimized EDFG is shown in the Fig. 11B. The signal R and respective variable is optimized. It is shown in the diagram with dashed lines and grey filling. The table in point C depicts in details the recursive generation process. It illustrates the node evaluation according to the methodology shown in the Fig. 9. Each operation node is numbered (indexed) in order to explicitly identify it. The generation process is performed for output and retained internal variables. Intermediate variables (associated with internal signals e.g. signal R) that are not used for storing process information between calculation cycles are removed. This is a part of an optimization process that enables removal of unused nodes of an EDFG.

The generation process starts from variable assignment node of the EDFG. According to the methodology shown in Fig. 9 argument nodes are recursively visited. Generation procedures nesting is shown in the table next to produced result in form of an IL code. The process starts from the assignment node of variable W and traces back to operation nodes. The first operation node is the AND node with index 3. The trace process selects arguments from vertical path (as shown in figure) traversing nodes 4 and 5 until reaching the variable A reading node. Both arguments of node 5 are delivered from variable read nodes. This enables generation of state-

ments shown in lines 1 and 2. Similar process is repeated for the node 4. The argument of node 3 is delivered by branch consisting of nodes 1 and 2 and respective variables. Entering this path generates the operation with parenthesis shown in line 3 that is completed after branch evaluation. Similarly to already described recursive generation process starting from the node 2 graph is back traced until node 1. The generation process produces sequence of instruction responsible for implementing operations described by nodes 1 and 2. The node 2 is referred by nodes 3 and 6. The result is immediately available for node 3 and must be preserved for evaluation of node 6. For this purpose the internal variable $V1$ is created. It should be recalled that variable R has been optimized. Immediately after leaving node 2 evaluation result is passed to node 3 that is shown in line 9 that completes the operation by closing the parenthesis. Finally the evaluation process is initiated for V variable. In the same way as for W variable reverse graph traversal is performed until the node 8. Starting from this point generation process is continued. The node 6 second argument is the node 2 that has already been evaluated and result was stored in an automatic temporary variable $V1$. The variable $V1$ is used instead of repeated evaluation of nodes 1 and 2. The internal variable management is shown in last column of the table with grey circles denoting value assignment and white circles denoting value access. The line that connects points denotes the variable lifetime. The moment of fetching $V1$ value is the last access to it. The variable $V1$ is released for new value assignment (variable lifetime line ends at access point).



Recursive IL code generation process Call stack	IL Line	Instructions		Variables allocation	
				V1	-
node(3) ⇨ arg_1(4) ⇨ node(4) ⇨ arg_1(5) ⇨ node(5) ⇨ arg_1(A)	1	LD	A		
node(3) ⇨ arg_1(4) ⇨ node(4) ⇨ arg_1(5) ⇨ node(5) ⇨ arg_i(B)	2	ANDN	B		
node(3) ⇨ arg_1(4) ⇨ node(4) ⇨ arg_i(W)	3	OR	W		
node(3) ⇨ arg_i(2)	4	ANDN	(		
node(3) ⇨ arg_i(2) ⇨ node(2) ⇨ arg_1(1) ⇨ node(1) ⇨ arg_1(E)	5	LD	E		
node(3) ⇨ arg_i(2) ⇨ node(2) ⇨ arg_1(1) ⇨ node(1) ⇨ arg_i(F)	6	ANDN	F		
node(3) ⇨ arg_i(2) ⇨ node(2) ⇨ arg_i(G)	7	OR	G		
	8	ST	V1		
node(3) ⇨ arg_i(2)	9		)		
	10	ST	W		
node(6) ⇨ arg_1(7) ⇨ node(7) ⇨ arg_1(8) ⇨ node(8) ⇨ arg_1(C)	11	LD	C		
node(6) ⇨ arg_1(7) ⇨ node(7) ⇨ arg_1(8) ⇨ node(8) ⇨ arg_i(D)	12	ANDN	D		
node(6) ⇨ arg_1(7) ⇨ node(7) ⇨ arg_i(V)	13	OR	V		
node(6) ⇨ arg_i(2)	14	ANDN	V1		
	15	ST	V		

Fig. 11. An example of complete PLC code generation process

3.4. EDFG based program decomposition. The standard program is obtained in two steps from the LD description. Initially it is converted into EDFG representation. An instruction stream is generated from intermediate form after optimization. The two steps generation process improves obtained result in several aspects. The translation method optimizes a program by reducing the logic and arithmetic expressions, dead code elimination and internal variables optimization. The problem of extracting processing conditions are model dependent [20]. The EDFG models calculations and enables formal analysis of dependencies for each variable assignment. Let the set S_f called function support, consist of variables for which positive and negative cofactors are different:

$$
\begin{aligned}
S_f &= \bigcup_{i=1}^{n} x_i : f_{x_i} \neq f_{\overline{x_i}}, \\
f_{x_i} &= f\left(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n\right), \\
f_{\overline{x_i}} &= f\left(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n\right).
\end{aligned} \tag{12}
$$

The set S_f is determined by reverse direction graph traverse until variable read nodes. This property allows for automatic program partitioning for event driven processing. The program partitioning process extracts independent tasks $S_f \cap S_g = \emptyset$ and also allows to determine mutual dependencies for creating variables watched set.

The task partitioning is closely connected with abilities of variable changes detection. The trigger set T_f for function f is defined as:

$$
\begin{aligned}
T_f &= \bigcup_{i=1}^{n} t_i : t_i \in S_f, \\
T_f &\supseteq S_f.
\end{aligned} \tag{13}
$$

The trigger item variable t_i is an atomic item that system is able to observe or calculate and is given as follows:

$$
t_i = \bigcup_{j=1}^{m} \left(x_{ta(i,j),n} \oplus x_{ta(i,j),n-1}\right), \tag{14}
$$

where *ta*(i, j) is a function that maps the variables to trigger variable t_i, n denotes the current calculation cycle, m declares the number of variables an elementary trigger item consists of. There the following remarks about trigger variables can be made. The ideal case is the one to one correspondence between the trigger item t_i and the x_i variable. Unfortunately, the trigger item incorporates several variables due to implementation limitation and processing overhead trade off. The trigger variables calculation can be implemented as a pure software or a hardware assisted decision block. The hardware implementation reduces the programming overhead and allows for calculating the trigger conditions in parallel with normal operations. The trigger calculation requires additional storage for value comparison coming from previous and current cycle. In case of software implementation this introduces requirement of additional memory for storing the previous value for comparison purposes. This overhead is eliminated by hardware assisted implementation that takes benefit from simultaneous read and write access to memory. To satisfy a multiple cycle variable changes propagation that has been exemplified in 3.1 all variables in EDFG with read access are monitored for changes.

4. Hardware assisted event detection

The event driven processing implemented with the use of programmatic approach introduces an overhead related to trigger calculation and requires additional storage space for difference tracking. The significant performance benefits can be achieved by integrated hardware-software co-design. Integrating variables changes tracking into a hardware significantly reduces the programmatic overhead and memory requirements. For specific implementation and evaluation purposes the FPGA devices are used. The unit is implemented as a part of the controller core [21].

The memory content difference detection is implemented without introducing any overhead in memory access (Fig. 12). The difference observer circuit is fully transparent for the system. The hardware assisted difference observation facilitate meeting the constraints and performance requirements for event driven system given by (6–11). In order to enable comparison between a current memory cell content and a written value the memory module with separated input and output buses is used with an edge triggered data write (sometimes called an addressable register). The difference detector accumulates observed differences since releasing an INIT signal.

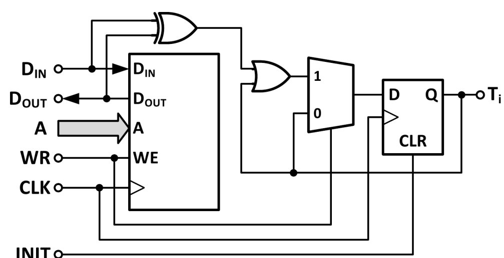


Fig. 12. The memory content transparent difference detector

The presented memory content observer guards entire space of a memory block. It does not allow to implement functionality described by (12–14) that forms guarded area for program block. In order to meet given requirements the unit shown in Fig. 13 has been designed. The figure depicts the guarded memory and an instance of difference detector unit. A decomposition process is applied that transforms the memory address into guarded regions identifiers marked as TA. This allows to reduce the number of address lines observed by aggregation LUT. Finally inside the guard detector the desired areas are observed according to the aggregation LUT memory content. Both membership and aggregation memories contents do not change at run time. They are programmed with guarded areas configuration data obtained at program compilation. The aggregation result is stored in a buffered register. The INIT input is activated at the beginning of the calculation cycle after input updates. It transfers a trigger flag from working register to the output register. The working register is initialized and a trigger calculation starts over guarding entire calculation cycle. For the first calculation cycle the trigger register is initialized with 1 requesting obligatory calculation process in first processing cycle.

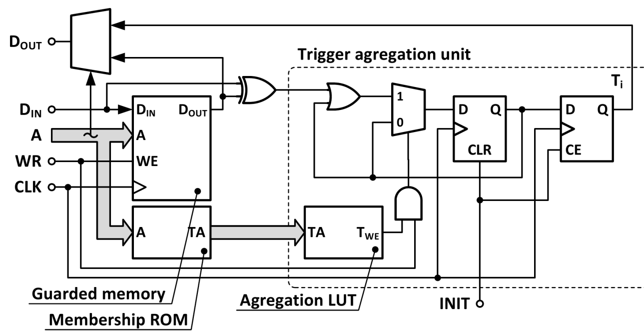


Fig. 13. The configurable difference detector

The guard unit has been implemented in Spartan FPGA families S3, S3E and S6. There has been considered two configuration of the unit with 64 or 256 programmable areas. The detailed resource utilization is shown in Table 1. The simple combinatorial path of unit does not hinder memory block operation. This is extremely important as the unit must be transparent for a PLC CPU operation. The event driven program execution requires a set of guard units. They are accessed as memory cells. Trigger flags are gathered in marker area space enabling PLC CPU access to them.

Table 1

Type	FPGA	LUT		FF	f_{MAX} [MHz]
		Logic	RAM		
G64	S3	19	4	2	219
	S3E	19	4	2	232
	S6	14	1	2	491
G256	S3	40	16	2	184
	S3E	32	16	2	181
	S6	16	4	2	407

5. Event driven program implementation

In order to illustrate substantially benefits coming from the event driven program execution an exemplary program has been selected. The program in form of the LD and its IL equivalent are shown in Fig. 14. The program consists from two control blocks mutually synchronized by B_1 and B_2 signals. The standard program consists of 22 instructions. Let assume a unit execution time for each instruction. The program does not contain conditional paths and it is executed uniformly in 22 time units.

The program can be accommodated for an event driven handling by splitting it into 4 groups according to driven variables. It should be noticed that selective execution inserts additional instructions for conditional evaluation of program blocks dependant of trigger variables state. The conditional block boundaries are determined by output variables assignments. This partitioning does not meet requirements of event driven processing while blocks for W_1 and W_2 signals doesn't meet instructions ratio between block and conditional entry. The output signals are merged into sets $\{V_1, W_1\}$ and $\{V_2, W_2\}$ with maximal number of common signals. The additional parts that are added to the program are shown in column for event driven blocks. There are defined two trigger variables $T1$ and $T2$ that controls evaluation of each block. Those

variables comes from configurable difference detectors. The places in standard program where are inserted the blocks for conditional execution are marked with grey rectangles. The program with selective execution adds 4 extra instructions that enable conditional execution of the blocks.

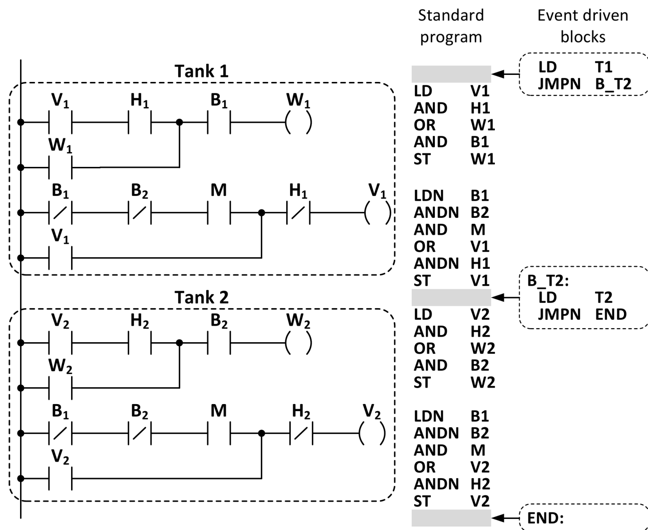


Fig. 14. The LD program implementation with event driven processing

5.1. The event driven program implementation performance. The event driven program is longer than its standard implementation but contains blocks that are executed conditionally. Schematically the calculation time is shown in the Fig. 15. There are marked two blocks evaluating V_1, W_1 and V_2, W_2 respectively. In Fig. 15A there is shown the standard execution approach. On the time chart there are marked maximal and minimal calculation times. In Fig. 15B and Fig. 15C are shown exemplary calculations for event driven processing. There are marked grey rectangles T_1 and T_2. They depict conditional entry to respective processing blocks The execution time for stable input signals is only 4 instructions where only blocks T_1 and T_2 are executed in loop. When changes are observed respective parts of program are included in execution process. It can be noticed that a signal change of B_1 or B_2 triggers evaluation of both blocks constituting the worst case calculation scheme shown in Fig. 15B. The average calculation time for the standard approach is 33 instructions while for event driven processing does not exceed 28 instructions. The worst case calculation time is 44 to 30 cycles instructions. The typical operation scheme is given in Fig. 15C. Only one calculation block is triggered. In such case the times.

The event driven approach response time improvement is observed with a growing number of conditionally executed blocks of infrequently changing signals. Than the decision block constitutes only a small fraction of processing blocks evaluation time. When there are two instances of the exemplary processes implemented in a program the average calculation time is 66 to 34 and maximal calculation time 88 to 38. An example of industrial object that is controlled with program, that evaluation time exceeds an acceptable emergency response is described in [10]. The event driven approach enables splitting a long program into blocks that are evaluated with signal changes arrival. This significantly reduces a response time assuring a fast response to an emergency request.

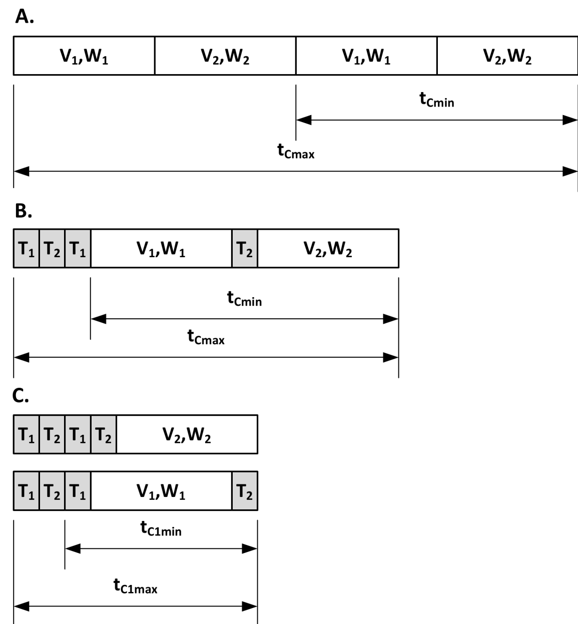


Fig. 15. Comparison of calculation times

6. Summary

The paper presents the selective program processing method for the PLC. It is focused on significant reduction of response time by eliminating redundant calculations. It brings entire a solution starting from a theoretical background of the processing concept with a performance evaluation and its limitations. Then it moves to an intermediate program representation with Enhanced Data Flow Graphs (EDFG). This part shows the concept of translating the control program given with the LD to the data flow graph. The graph representation is used for optimisation of a program. The next step describes the code generation method from an EDFG. There is shown the iterative algorithm for a code generation together with temporary variables allocation. Finally, the EDFG is used for an automatic program partitioning enabling selective processing. The developed system takes benefits from hardware units that detect differences in a memory content. The unit is completely transparent for memory operations. The memory guarded area is fully programmable. The proposed decomposition of a membership function of the guarded area reduces a hardware complexity and propagation time of the difference detector unit.

The research process over an efficient control program implementation is ongoing. The FPGA technology enables quick evaluation of complex systems and solutions. The next step that is moving an event driven processing further is a systematic method of passing entire the scheduling process of tasks to hardware unit that will select tasks for execution. This will enable to keep easy programmability with the highest performance of the system obtained through the developed program

analysis and synthesis methods eliminating execution overhead.

Acknowledgements. This work was supported by the Ministry of Science and Higher Education funding for statutory activities (decision no. 8686/E-367/S/2015 of 19 February 2015).

REFERENCES

[1] Cenelec, *EN 61131-3, Programmable Controller – Part 3: Programming Languages*, Int. Standard Management Centre, Brussels, 2013.

[2] M. Chmiel and E. Hrynkiewicz, "Concurrent operation of processors in the bit-byte CPU of a PLC", *Control and Cybernetics* 39 (2), 559–579 (2010).

[3] M. Chmiel, "On reducing PLC response time", *Bull. Pol. Ac.: Tech.* 56 (3), 229–238 (2008).

[4] J. Klamka, "Controllability of dynamical systems. A survey", *Bull. Pol. Ac.: Tech.* 61 (2), 335–342 (2013).

[5] T. Klopot, P. Laszczyk, K. Stebel, and J. Czeczot, "Flexible function block implementation of the balance-based adaptive controller as the potential alternative for PID-based industrial applications", *Trans. Institute of Measurement and Control* 36 (8), 1098–1113 (2014).

[6] G. Valencia-Palomo and J.A. Rossiter, "Programmable logic controller implementation of an auto-tuned predictive control based on minimal plant information", *ISA Trans.* 50, 92–100 (2011).

[7] K.H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*, Springer-Verlag, Berlin, 2010.

[8] K.H. Koo, G.S. Rho, and W.H. Kwon, "An architecture of the RISC processor for programmable controllers", *20th Int. Conf. on Industrial Electronics, Control and Instrumentation* IECON 94, 1179–1183 (1994).

[9] S.A. Edwards, K. Sungjun, E.A. Lee, I. Liu, H.D. Patel, and M. Schoeberl, "A disruptive computer design idea: architectures with repeatable timing", *IEEE Int. Conf. on Computer Design* 1, 54–59, (2009).

[10] M. Chmiel, A. Malcher, and A. Nowara, "A metal sheet etching process control", *Machines, Technology, Materials* 2, CD-ROM (1997), (in Polish).

[11] A. Falcione and B.H. Krogh, "Design recovery for relay ladder logic", *IEEE Control Systems* 13 (2), 90–98 (1993).

[12] D. Du, X. Xiaodong, and Y. Kazuo, "A study on the generation of silicon-based hardware PLC by means of the direct conversion of the ladder diagram to circuit design language", *Int. J. Advanced Manufacturing Technology* 49 (5), 615–626 (2010).

[13] J. Mocha and D. Kania, "Hardware implementation of a control program in FPGA structures", *Przegląd Elektrotechniczny* 88 (12a), 95–100 (2012).

[14] A. Milik and E. Hrynkiewicz, "On translation of LD, IL and SFC given according to IEC-61131 for hardware synthesis of reconfigurable logic controller", *IFAC World Congress* 19, 4477–4483 (2014).

[15] Y. Yi and C. Haidan, "An optimizing compiler method to avoid partial invalid PLC instructions", *IEEE Int. Symp. on Industrial Electronics* 1, 80–83 (2010).

[16] A. Milik and A. Pulka, "On FPGA dedicated SFC synthesis and implementation according to IEC61131", *Int. Conf. on Signals and Electronic Systems* 1, CD-ROM (2014).

[17] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, New York, 1976.

[18] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms,* Kluwer Academic Publisher, Boston, 1996.

[19] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, 1994.

[20] H. Souleiman, S. O'Riain, and E. Curry, "Approximate semantic matching of heterogeneous events", *ACM Int. Conf. on Distributed Event-Based Systems* 1, 252–263 (2012).

[21] M. Chmiel, E. Hrynkiewicz, J. Mocha, and A. Milik, "Central processing units for PLC implementation in Virtex-4 FPGA", *IFAC World Congress* 18, 6902–6907 (2011).