IEICE TRANS. FUNDAMENTALS, VOL. E78-A, NO. 11 NOVEMBER 1995

PAPER Special Section on Net Theory and Its Applications to Discrete Event System Design

Petri Nets-Based Super Scalar Computing in Programmable Controllers

Naehyuck CHANG†, Jaehyun PARK††, *and* Wook Hyun KWON†, *Nonmembers*

SUMMARY This paper proposes a hardware architecture of programmable controller based on Petri nets. The suggested architecture achieves sufficiently rapid processing even as demands on PCs become increasingly more complex. The architecture's speed and efficiency are derived from an automatic and dynamic super scalar computing capability that executes bit instructions and data handling instructions simultaneously without preprocessing, due to the properties of Petri nets. Specific characteristics for both architectural memory-based implementation of Petri nets and evolution algorithms are suggested and classified by the net structure. Analysis of the suggested architectures and effects on performance are also given with mathematical formulas and a computer simulation.

key words: *programmable controller, sequence control, super scalar, Petri nets*

1. Introduction

Programmable controllers (PCs) are widely used in factory automation. These PCs which employ a *discontinuous* or *on/off* control are very important for *sequential control systems* because many machines consist of *units* controlled by a large number of *sequence step.* Although many *continuous controls* are used inside of each unit, in terms of hierarchical view, upper levels of the automation are controlled by a sequence control systems [9].

As the demands on PCs become more rigorous due to today's increasingly more complex and larger factory automation, higher performance programmable controllers are required. This higher performance can be achieved by increasing instruction execution speed. Three kinds of instruction are available in an application program of a PC. The *bit instruction* consists of boolean functions to emulate *mechanical relays.* The *data handling instruction* consists of *move, compare, BCD-BIN converion* and many *arithmetic instructions.* The data handling instructions perform a more complex control than the bit instruction does. Due to these two complex instructions, the PCs can execute more complex operations called *special instruction* such as *PID loop instruction* [9].

The major portion of instructions in application programs are bit instructions, and because the bit instruction is a simple boolean operation, the execution time is below 100 nsec per step in the case of a high performance PC. On the contrary, the execution time of an data handling instruction is relatively slower than that of the bit instruction. So to speed up a PC, both instructions must be executed quickly.

To execute both the bit and the data handling instructions rapidly, most high performance PCs have separate dual functional units for the bit and the data handling instructions respectively. To maximize the performance with such dual functional units, a *super scalar* computing technique can be used. Importantly, super scalar computing is feasible for most commercial computer systems. In a scientific calculation problem, super scalar computing is usually performed between integer instructions and floating point instructions, and they do not have tight data dependency with each other. In the case of an application with tight data dependency, computers based on Von Neumann structure can hardly support super scalar computing without intensive off-line preprocessing. Even with the highly optimized compiler, super scalar computing is not an easy operation to implement.

Super scalar computing in PCs occurs when the bit functional unit and the data handling functional unit are used simultaneously. This simultaneity poses a problem because every data operand in PC programs is highly dependent on the other data operands [2]. Moreover, the bit data and word data are tightly combined together. For this reason, commercial PCs using Von Neumann structure do not support the super scalar computing, in spite of the separated dual functional units.

Moreover, an optimized compilation for super scalar computing is not adequate for PCs for the following reason. Programming the PCs is usually *field-programming.* Fast and easy compilation is very important feature to adjust the control system in the field. Therefore the programming sequence must be simple. In most cases, compiling a *ladder diagram (LD)* program in binary codes is a *substitution* not a *compilation* for fast and easy programming. And sometimes *back annotation* is also required, which generates a LD program from binary codes.

To overcome the above problems, a new algorithm and an architecture which performs super scalar

Manuscript received March 31, 1995.
Manuscript revised June 14, 1995.

†The authors are with the Dept. of Control and Instrumentation Engr., Seoul Nat'l Univ., Korea.
††The author is with the University of Michigan, Real-time Computing Laboratory, U.S.A.

computing using a LD program is required. Super scalar computing should not require intensive preprocessing, either. An on-line resolving of the data dependency problem can be carried out using the data flow model [2]. Petri nets are useful to model data dependency relations, and their parallelism is useful to super scalar computing. To develop a PC hardware architecture based on concurrency control semantics of Petri nets is a good solution to the above problems.

In this paper, an on-line optimization of evolution sequence for super scalar computing in PCs is proposed. The proposed method is dynamic and automatic [7], [8]. First, PCs are programmed in LD, like commercial PCs. Second, the data flow model of the LD program is captured. The data flow model is described in Petri nets, and the capturing process is almost an one-to-one conversion [1]. The third and last step is to develop a hardware architecture design based on the Petri net model. The implementation method is a memory-based implementation [5], which is fast, regular, and flexible. To achieve a large size Petri net, a binary Petri net model is also adopted [1], [4].

In Sect. 2, the problem of super scalar computing in PCs is introduced. In Sect. 3, a hardware architecture and mathematical description of the suggested Petri net-based PC are described. Section 4 is an analysis of the suggested PC. In Sect. 5, super scalar computing in the suggested Petri net-based PC is estimated and quantified with particular emphasis on the performance enhancement as revealed through computer simulation.

2. Super Scalar Computing in Programmable Controllers

Most common PCs adopt the LD to their application programming language. High level languages such as *Sequential Function Chart (SFC)* are also used, but usually they are translated to the LD internally for evolution. Figure 1 shows some typical instructions of used in PCs. Because the execution methods of data handling instructions including special instructions are similar and they are based on arithmetic operations, for convenience they are called *arithmetic instruction* in this paper. As shown in Fig. 1, each arithmetic instruction is tightly combined to bit instructions, in terms of data flow. The LD is a kind of combinational logic consisting of many boolean functions. Sometimes they have *memory* elements such as *timers, counters, filp flops* and so on, but the portion is small.

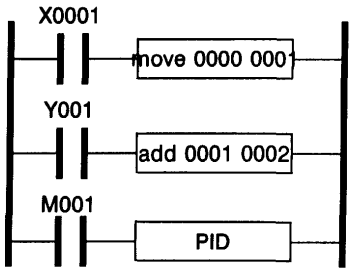

Fig. 1 LD example.

The frequency of use of bit instructions and arithmetic instructions varies according to the application, but many studies have been performed to design the hardware resources most efficiently [3]. Statistics from the real application programs show that a PC spends about three times longer in executing arithmetic instructions than it does in executing bit instructions. For this reason, high performance programmable controllers use a lot of hardware resources to execute arithmetic instructions as quickly as possible [2].

Though most high performance programmable controllers have separate units for bit functions unit and arithmetic functions, they cannot use the functional units simultaneously. In contrast, super scalar computing in programmable controllers enables the program sequencer to issue both bit and arithmetic instructions to their respective functional units simultaneously.

Moreover, with super scalar computing, fast execution time can be achieved without additional cost. The execution time of bit instructions and arithmetic instructions are defined as follows.

t_i^l : execution time of i-th bit instruction.

t_j^m : execution time of j-th arithmetic instruction.

where, $1 \leqq i \leqq L$, $1 \leqq j \leqq M$ and $L+M$ is the total number of instructions.

Suppose a particular PC has just one bit functional unit and one arithmetic functional unit. Without the super scalar computing, total execution time $\mathcal{T}$ is the summation of each instruction execution time, as the instructions must be executed one at a time. The total evolution time $\mathcal{T}$ without the super scalar computing is

$$\mathcal{T} = T^l + T^m. \tag{1}$$

With super scalar computing (in the perfect case that all bit instructions and arithmetic instructions are executed simultaneously), total execution time $\mathcal{T}$ is the longest execution time among all the individual instruction execution times. The evolution time $\mathcal{T}$ with the perfect super scalar computing is

$$\mathcal{T} = \max(T^l, T^m), \tag{2}$$

where

$$T^l = \sum_{i=1}^{L} t_i^l, \tag{3}$$

$$T^m = \sum_{j=1}^{M} t_j^m. \tag{4}$$

In fact, the perfect case is highly unlikely because of a data dependency problem. Therefore actual evolution

time $\mathcal{T}$ is

$$\max(T^l, T^m) \leqq \mathcal{T} \leqq T^l + T^m. \quad (5)$$

In this paper, the work to make $\mathcal{T}$ closer to $\max(T^l, T^m)$ is described. The data dependency problems can be resolved using the data flow graph. Using the data flow graph, an instruction of the specific operands that are ready can be dynamically and automatically scheduled without a preprocessing[2]. The data flow graph can also be described using Petri nets. In the case of Petri nets, due to parallelism, the data dependency problem can be resolved. Therefore to approach the goal, a Petri nets-based programmable controller is suggested.

3. A Petri Net-Based Programmable Controller

The proposed programmable controller consists of *control circuits* based on Petri nets, *functional units (FUs)*, and *I/O controller* as shown in Fig. 2. In the case of a Von Neumann-structured programmable controller, the control circuits consist of a *sequencer*, a *program memory*, a *data memory* and so on. Unlike those found in a Von Neumann-structured programmable controller, the FUs in a Petri net-based PC are controlled by the *memory-based implementation of Petri nets* based on a data flow control scheme to resolve the data dependency problem. The suggested programmable controller is described as follows.

The Petri net-based programmable controller Θ is defined as 9-tuples,

$$\Theta = (P, T, I, O, \mu, \mu_0, F, D, \Gamma), \quad (6)$$

where

$P = P^X \cup P^M \cup P^Y$, $P^X \cap P^Y = \emptyset$,

$P^Y \cap P^M = \emptyset$, $P^M \cap P^X = \emptyset$,

$P^X = \{p_1^X, p_2^X, ..., p_l^X\}$ is a set of sensor input places,

$P^M = \{p_1^M, p_2^M, ..., p_m^M\}$ is a set of internal relay places,

$P^Y = \{p_1^Y, p_2^Y, ..., p_n^Y\}$ is a set of actuator output places,

$T = \{t_1, t_2, ..., t_k\}$ is a set of transitions,

$I(t) = \{p|(p,t) \in F\}$ is the set of input places of t,

$O(t) = \{p|(t,p) \in F\}$ is the set of output places of t,

$O(p) = \{t|(p,t) \in F\}$ is the set of output transitions of p,

$\mu = (\mu(p_1), ..., \mu(p_{l+m+n}))$ is $(l+m+n)\times 1$ vector, markings of places,

$\mu_0 = (\mu_0(p_1), ..., \mu_0(p_{l+m+n}))$ is an initial marking,

$F(t) = \{f|(t,f)\}$ is a set of instructions defined at t,

$D(p) = \{d|(p,d)\}$ is a set of data defined at p,

$\Gamma = (\tau_1, ..., \tau_k)$ is $k \times 1$ vector, execution time of transitions.

And $F \subseteq (P \times T) \bigcup (T \times P)$ is a set of all arcs of Θ.

The data flow structure of the application program is designed based on behavioral semantics of Petri nets[1], and it can be easily captured from a LD program[2]. Figure 3 is a translated data flow model of Fig. 1. Since the data flow model is a sub-class Petri net model, it can be described in Petri nets[10],[11].

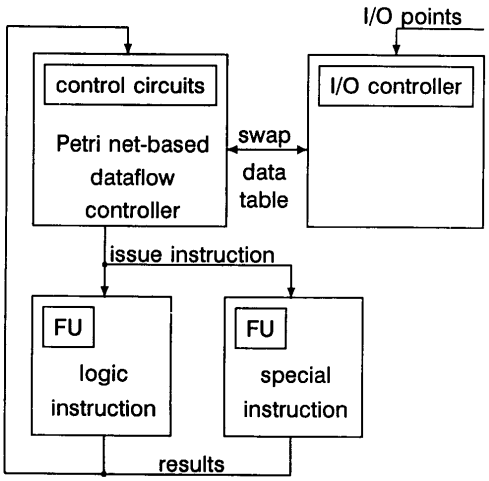

Fig. 2 An architecture of Petri net-based programmable controller.

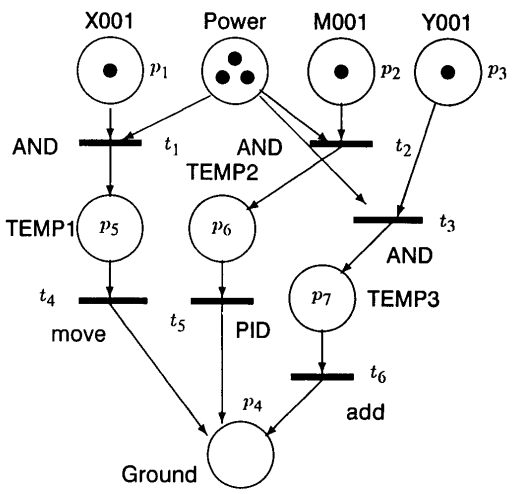

Fig. 3 A data flow conversion and a Petri net model example.

Moreover, if the LD program has no data dependent branches, it can be described in marked graphs also, and in most cases LD programs can be coded with no data dependent branches[1]. The following section describes the properties of the suggested Petri net-based programmable controller.

4. Basic Algorithms and Architectures

In the case of ordinary Petri nets, a single *place* can be an input place of more than two transitions and an output place of more than two transitions. Therefore, more than two elements of marking in μ and data in $D(p)$ may be mapped to a transition. But in the case of the marked graph, every element of marking in μ and data in $D(p)$ can be mapped to a transition once and for all. Therefore, a table of marking μ and data $D(p)$ can be contained in the transition table as shown in Fig. 4, and the required memory cycles are reduced from two cycles to one cycle. The marked graph has enough modeling power to describe a LD, and for this reason, the suggested PC adopts a marked graph that is a sub-class of Petri nets.

The marked graph can be classified depending on whether the marked graph is *fan out free*[6] or not. If the marked graph is not fan out free, the model will be converted to a binary-fan out marked graph, say, a binary-fork marked graph[1]. As basic algorithms, two kinds of evolution algorithms are suggested. In Algorithm 1, the next enabled transition is pulled out from the *queue*, and the output transition, which is the output place of the fired transition, is inserted to the *queue*. The *queue* contains start places that have initial markings.

Various detailed structures of the control circuit block shown in Fig. 2 and FUs are suggested in Figs. 5, 6, 8 and 9. MUX signifies a 2-to-1 transition multiplexer with register, TBL signifies a table of $D(I(t))$, $\mu(I(t))$ and $F(t)$ of which index is transition, and FF signifies a flip flop to indicate whether the current transition is being fired or not. The queue symbol in the figures contains $O(p)$ tables, and FU signifies a common functional unit built in common microprocessors.

Algorithm 1:

step 1 For an initial state, insert the initial places to the *queue*.

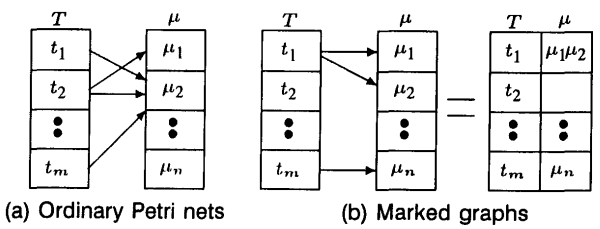

Fig. 4 Relations between transition T, marking μ and data $D(p)$.

step 2 Get next transition $t_j = O(p_i)$ from the *queue*.

step 3 Get $O(O(t_j))$,

Test if $\forall p \in I(t_j)$, $\mu(p) = 1$,
then go to step 4 (fire),
else go to step 2 (no fire).

step 4 Insert $O(t_j)$ to the *queue* and go to step 2.

In Algorithm 2, the next enabled transition is pulled out from the output transition, which is an output place of the current fired transition, repeatedly until the next transition is not enabled. If the next transition is not enabled, the new next transition will be pulled out from the *queue*.

Algorithm 2:

step 1 For an initial state, insert the initial places to the *queue*.

step 2 Get next transition $t_j = O(p_i)$ from the *queue*.

step 3 Get $O(O(t_j))$,

Test if $\forall p \in I(t_j)$, $\mu(p) = 1$.
then go to step 3 (fire),
else go to step 2 (no fire).

A minimum hardware structure to evolve fan out free marked graphs according to Algorithm 1 is shown in Fig. 5. Because the marked graphs are fan out free, only one next transition should be inserted to the *queue* at a firing. Another hardware structure to evolve fan out free marked graphs is shown in Fig. 6. The hardware is to minimize *queue* access, and no write operations are required. Because of the fan out free condition, Algorithm 1 and 2 require the same evolution time. The enabled transition means an instruction of which operands have been already resolved, in the case that the operands are the previous instruction's result. If the current transition is not enabled, the hardware will examine another transition. This shows on-line resolving of the data dependency problem. Lemma 1 shows execution time of simple memory-based hardware.

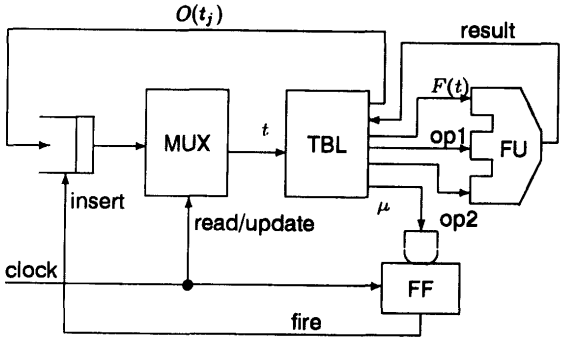

Fig. 5 Fan out free with Algorithm 1.

Lemma 1: Let the propagation delay or access delay time of each device be determined as follows.

t_{qr} : access delay time of the queue read operation.

t_{qw} : access delay time of the queue write operation.

t_{tr} : access delay time of TBL (table) read operation.

t_{tw} : access delay time of TBL write operation.

t_f : propagation delay time of FU (functional unit).

t_m : propagation delay time of MUX (multiplexer).

t_t : propagation delay time of FF (Firing control Flip flop).

Then the evolution time t_{ev}^{a1} of one firing of Algorithm 1 and t_{ev}^{a2} of one firing of Algorithm 2 are,

$$t_{ev}^{a1} = \max(t_{qr}+t_{qw}, t_{tr}+\max(t_f, t_t)+t_m+t_{tw}), \tag{7}$$

and

$$t_{ev}^{a2} = \max(t_{qr}, t_{tr} + \max(t_f, t_t) + t_m + t_{tw}) \tag{8}$$

respectively.

Proof: In Fig. 7, synchronization of the hardware in Fig. 5 and Fig. 6 is shown. Operations which can be carried out simultaneously are represented as parallel lines. For the synchronization, the aforementioned evolution time must be required. □

Memory devices in Fig. 5 and Fig. 6 are FIFO queue and RAMs. Without loss of generality,

$$t_{qr} \approx t_{qw} \approx t_{tr} \approx t_{tw} \tag{9}$$

is assumed, and

$$t_m \approx t_t \approx t_f \ll t_{qr} \tag{10}$$

is also assumed in the case of the bit functional unit.

However not all real application programs of programmable controllers are fan out free. For this reason, the hardware must evolve marked graphs that are not fan out free. And the fan out number of the fan out points [6] vary to their applications. To overcome this problem, multiple fan out points are converted to cascaded *binary-fan out* points, called *binary-fork conversion* [1]. Each binary-fan out point subsequently generates two different transitions at a firing. To evolve the binary-fan out marked graphs by Algorithm 1, single *queue* read access and double *queue* write accesses are required. In this case, for Algorithm 1,

$$t_{ev}^{a1} = \max(t_{qr} + 2t_{qw}, t_{tr} + \max(t_t, t_f) + t_m + t_{tw}), \tag{11}$$

and for Algorithm 2,

$$t_{ev}^{a2} = \max(t_{qr}+t_{qw}, t_{tr}+\max(t_t, t_f)+t_m+t_{tw}). \tag{12}$$

Therefore the evolution speed of Algorithm 1 is slower than that of Algorithm 2 in the case of binary-fan out marked graphs. A hardware structure of binary-fan out marked graph with the evolution of Algorithm 2 is shown in Fig. 8. Algorithm 3 is a modified version of Algorithm 2 to cope with the binary-fan out.

Algorithm 3:

step 1 For an initial state, insert the initial places to the *queue*.

step 2 Get next transition $t_j = O(p_i)$ from the *queue*.

step 3 Get $O(O(t_j))$,

Test if $\forall p \in I(t_j)$, $\mu(p) = 1$.

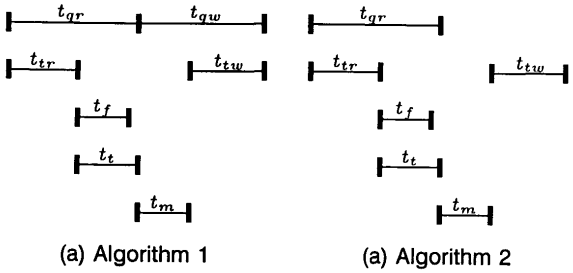

Fig. 7 Evolution time according to Algorithm 1 and Algorithm 2.

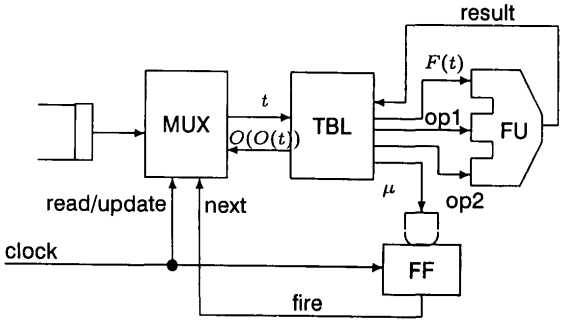

Fig. 6 Fan out free with Algorithm 2.

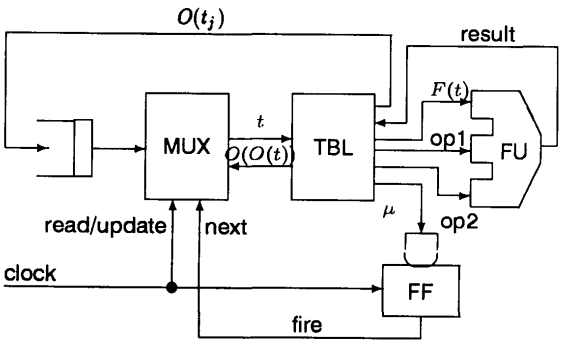

Fig. 8 Binary-fan out with Algorithm 2.

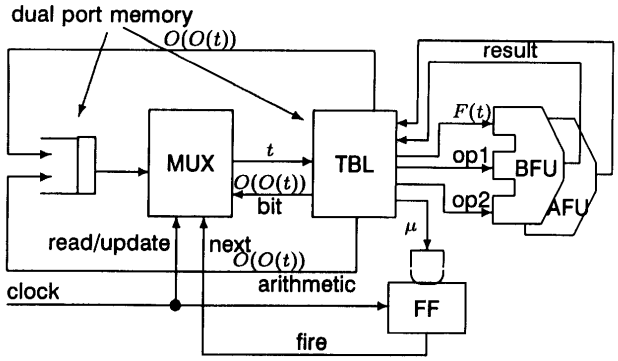

Fig. 9 Structure for the super scalar computing.

then go to step 3 (fire) and
if fan out point insert
$O(t_j)$ to the *queue*,
else go to step 2 (no fire).

5. The Structure of Super Scalar Computing

In super scalar computing, the enable test of the next transition should be initiated, even if the current operation is not finished. Therefore Algorithm 2 and Algorithm 3 cannot be used in the case of the super scalar computing because the next transition is the output transition of the current output place. Therefore, for super scalar computing, Algorithm 1 should be adopted in case of the binary-fan out, though the evolution speed will be degraded. To overcome this speed degradation, a choice of algorithms must be available. Thus, Algorithm 4 is suggested for super scalar computing, where Algorithm 3 is adopted in the case of bit instructions and Algorithm 1 is adopted in the case of arithmetic instructions. The binary-fan out point should be the bit instruction only, for simplicity. The hardware can test the next transition after issue of the arithmetic instruction, which is not finished yet. The architecture has two functional units, a *bit function unit (BFU)* for bit instructions and an *arithmetic function unit (AFU)* for arithmetic instructions, as shown in Fig. 9. If the current instruction is an arithmetic instruction and the operands are resolved, the instruction will be issued. And immediately after the issue, the next instruction is examined regardless of the finish of the instruction automatically. Therefore dynamic super scalar computing is achieved. In the case of the AFU,

$$t_m \approx t_t \ll t_{qr} < t_f, \tag{13}$$

so during the execution of the arithmetic instruction, many bit instructions can be issued. The performance of the suggested super scalar computing architecture is analyzed using a computer simulation, according to the following conditions and assumptions. To execute a bit instruction, one *cycle time* is required. one cycle time

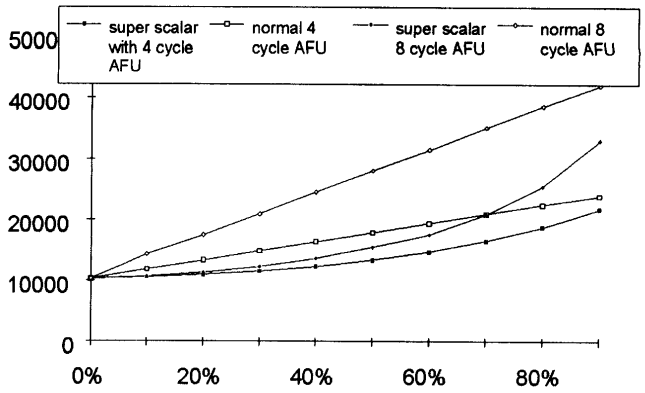

Fig. 10 Comparison of evolution time between super scalar computing and normal computing.

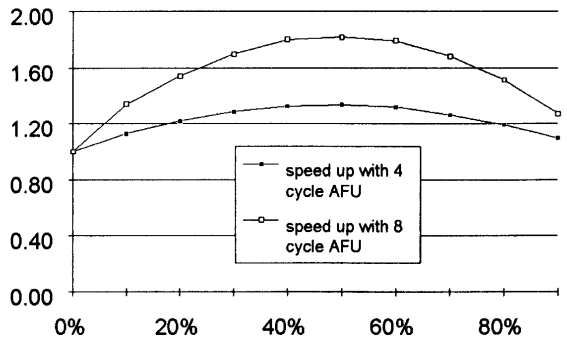

Fig. 11 Speed up of the suggested super scalar computing.

t_{cycle} is

$$t_{cycle} = \max(t_{qr} + t_{qw}, t_{tr} + t_{BFU} + t_m + t_{tw}), \tag{14}$$

where t_{BFU} is the propagation delay of the bit functional unit. In the case of an arithmetic instruction, two cases are considered. One is $t_{AFU} = 4$ cycles, and the other is $t_{AFU} = 8$ cycles, where t_{AFU} is the propagation delay of the arithmetic functional unit. Of course, if the AFU is not empty, another arithmetic instruction cannot be issued. The probability that a transition is enabled is 50%. The total step of instruction is 5,000. The arithmetic instructions are randomly scattered. The portion of the arithmetic instructions is the horizontal axis in both of Figs. 10 and 11. Figure 10 shows the evolution times in the number of cycles. In the figure, the evolution time of super scalar computing is compared to the evolution time of normal computing. The performance enhancement due to super scalar computing is shown in Fig. 11.

Algorithm 4:

step 1 For an initial state, insert the initial places to the *queue*.
step 2 Get next transition $t_j = O(p_i)$ from the *queue*.
step 3 Get $O(O(t_j))$,

Test if $\forall p \in I(t_j)$, $\mu(p) = 1$,

then
if bit instruction go to step 3 (fire) and if fan out point insert $O(O(t_j))$ to the queue.
if arithmetic instruction go to step 2 (fire and next issue) and insert $O(O(t_j))$ to the *queue.*
else if go to step 2 (no fire).

6. Conclusion

In this paper, a Petri net-based programmable controller architecture is proposed. The main objective is to make a programmable controller with a super scalar computing capability, because commercial Von Neumann-structured programmable controllers do not support the super scalar computing capability even with separated dual functional units.

To achieve the automatic and dynamic super scalar computing that is suitable to a field programming environment with a ladder diagram application program, Petri nets can be used in controlling the functional units. Four hardware architectures and algorithms are suggested, from the basic computing hardware to the super scalar computing hardware. Furthermore to implement a hardware that can evolve large size Petri net models with practical memory size, a binary-fan model is used. Importantly, the performance enhancement due to the super scalar computing is impressive, as show in in a computer simulation.

The suggested programmable controller architecture uses Petri nets based on data flow control scheme. At an application level, however, a ladder diagram can be used. The ability of the suggested programmable controller architecture to accommodate a ladder diagram is very important because it is a widely used language in the programmable controller area. The suggested architecture maximizes PC performance without additional costs for fast functional units.

References

[1] N. Chang, J. Park, G.S. Rho, and W.H. Kwon, "Memory-based implementation of a binary-marked graph with applications to programmable controllers," IEEE Trans. Industrial Elec. submitted.
[2] J.Park, N. Chang, G.S. Rho, and W.H. Kwon, "Implementation of parallel logic solving algorithm for PLC based on data flow architecture," IFAC Control Engineering Practice, vol.1, no.4, Aug. 1993.
[3] G.S. Rho, K.-H. Koo, N. Chang, J. Park, Y.-G. Kim, and W.H. Kwon, "Implementation of a RISC microprocessor for programmable controllers," Microprocessors and Microsystems. accepted.
[4] P.C. Baracos, R.D. Hudsin, L.J. Vroomen, and P.J.A. Zsombor-Murray, "Advances in binary decision based programmable controllers," IEEE Trans. Industrial Elec., vol.25, no.3, Aug. 1988.
[5] L.D. Coraor, P.T. Mulina, and O.A. Morean, "A general model for memory-based finite-state machines," IEEE Trans. Comput., vol.C-36, pp.175–184, 1987.
[6] S. Chakravarty, "A characterization of binary decision diagrams," IEEE Trans. Computers, vol.42. no.2, Feb. 1993.
[7] V.P. Srini, "An architectural comparison of data flow systems," IEEE Trans. Computers, pp.68–87, March 1986.
[8] I. Waston and J. Gurd, "A practical data flow computer," IEEE Trans. Computers, pp.51–57, Feb. 1982.
[9] I.G. Warnock, "Programmable Controllers, Operation and Application," Prentice Hall, 1988.
[10] T. Murata, "Petri nets: properties, analysis and applications," Proceedings of IEEE, vol.77, no.4, April 1989.
[11] J.L. Peterson, "Petri Net Theory and the Modeling of Systems," Prentice-Hall, Inc., 1981.

Naehyuck Chang was born in Korea on 19 Mar. 1967. He received BS and MS degrees in Dept. of control and instrumentation engineering from the Seoul Nat'l Univ. in 1989 and 1992, respectively. He is currently a PhD candidate in the same department in the Seoul Nat'l Univ. His research interests are discrete event system, real-time system and computer applications for a factory automation.

Jaehyun Park received BS, MS and PhD degrees in control and instrumentation engineering department from Seoul National University in 1986, 1988 and 1994 respectively. He is currently a research fellow of Electrical Engineering and Computer Science of the University of Michigan. He and his colleagues are currently building a mesh multicomputer, and an open architecture machining controller, called UMOAC. His main research interests are computer architecture for real-time control system, computer application for factory automation, and discrete event systems.

Wook Hyun Kwon received the B.S. and M.S. degrees in electrical engineering from Seoul National University, Seoul, Korea, in 1966 and 1972, respectively. He received a Ph.D. in control from Brown University in 1975. From 1975 to 1976, he was a research associate at Brown University. From 1976 to 1977, he was an adjunct professor at the University of Iowa. Since 1977, he has been with Seoul National University, now as a professor. From 1981 to 1982, he was a visiting assistant professor at Stanford University. Since 1991, he has been the director of Engineering Research Center of Advanced Control and Instrumentation which is being funded from Korean Science and Engineering Foundation. His main research interests are currently multivariable robust and predictive controls, discrete event system, automation network and computer applications for factory automation.