Parallelizing Factory Automation Ladder Programs by OSCAR Automatic Parallelizing Compiler

Tohma Kawasumi[1], Yuta Tsumura[1] Hiroki Mikami[1], Tomoya Yoshikawa[2,3], Takero Hosomi[2,4], Shingo Oidate[2,5], Keiji Kimura[1,6], and Hironori Kasahara[1,7]

[1] Waseda University,27 Waseda-machi, Shinjuku-ku, Tokyo, 1620042 {tohma, tmtmwaseda, hiroki}@kasahara.cs.waseda.ac.jp
[2] Mitsubishi Electric Corporation, Tokyo Building, 2-7-3, Marunouchi, Chiyoda-ku, Tokyo, 1008310
[3] Yoshikawa.Tomoya@aj.mitsubishielectric.co.jp
[4] Hosomi.Takero@ap.mitsubishielectric.co.jp
[5] Oidate.Shingo@dx.mitsubishielectric.co.jp
[6] keiji@waseda.jp
[7] kasahara@waseda.jp

Abstract. Programmable Logic Controllers (PLCs) and their programming language, or Ladder language, have been widely used for more than 50 years to control plants like Factory Automation or FA. Demands for higher performance of Ladder programs on PLCs are increasing along with increasing functionality and complexity of plants, as well as growing numbers and variety of sensors and actuators. Traditional clock frequency improvement of a CPU in a PLC is inappropriate to satisfy them since high reliability and robustness are essential for plant control, because of surrounding electrical noise. Instead, parallel processing on a multicore seems to be a promising approach. However, Ladder programs have poor loop parallelism and basic block level fine task granularity. This paper proposes a parallelization technique of Ladder programs by OSCAR automatic parallelizing compiler. It first translates a source Ladder program into an OSCAR compiler-friendly C program by a newly developed automatic translation tool. Then, the compiler parallelizes it. At the parallelization, the OSCAR compiler employs parallelism among macro tasks, each composed of a basic block in this application. However, the execution time of a basic block is relatively short compared with data transfer and synchronization overhead. Therefore, macro-task fusion is applied considering data dependency among macro tasks on a macro task graph so that the execution time of the fused macro task can be longer than the overhead and the parallelism among the fused macro tasks can be kept. Before the macro task fusion, the duplication of the basic block having a conditional branch and the graph transformation changing a macro task graph with control-dependence edges into a macro-task graph with just data dependence edges are applied. Finally, the macro tasks on the macro task graph having data dependence edges are statically scheduled on processor cores. A performance evaluation on two ARM Cortex A53 cores on a Zynq UltraScale+ MPSoC ZCU102 shows the proposed technique can reduce 17% of execution clock cycles, though a parallel program before the proposed task fusion needs twice longer execution time on two cores against a sequential execution.

Keywords: Ladder-program · Parallelizing compiler · Task fusion · Static scheduling.

1 Introduction

Instead of classic hardwired relay circuit-based sequence controllers, Programmable Logic Controllers (PLCs) consisting of CPUs have been widely used for plant control because of their flexibility, low maintenance cost, and small footprint. Among several PLC programming languages, Ladder language is a representable one. It can represent relay circuits and offers a low transition cost from classic sequence controllers. A Ladder program consists of two kinds of circuit blocks: condition parts and execution parts. Condition parts process boolean operations and check their results. Execution parts include operations, which are executed according to the results from condition parts.

Along with the advancement of plant control technology, demands for higher execution speed of Ladder programs are also increasing. Lower response time of PLC obtained by reduced Ladder program execution time allows more sensors and actuators resulting in precise target plant control. Further, recent plants' scale tends to become larger and more complicated, and their Ladder programs are also more extensive. This trend also introduces the motivation for faster Ladder program execution time.

A traditional approach of increasing the clock frequency of a CPU in a PLC is inappropriate because it makes keeping high reliability and durability difficult in a high electrical noise plant environment. Parallel processing of a Ladder program seems to be a promising approach to accelerate it. However, it usually has low loop parallelism and fine task granularity, resulting in the difficulty of employing conventional loop-level parallel processing and simple task parallel processing. Since a PLC is usually used in a severe environment, low heat dissipation realized by low power consumption is also expected.

Several parallel acceleration techniques for Ladder programs have been proposed. One focuses on logic operations in a program and reduces executed instructions. Another tries to parallelize Ladder programs. However, they are difficult to deal with indirect access by an index register, which a Ladder program characteristically uses. Program restructuring techniques for program acceleration are also challenging.

In contrast, we have developed OSCAR (Optimally SCheduled Advanced multiprocessoR) automatic parallelizing compiler [3–5]. It employs coarse grain task parallel processing and near-fine grain parallel processing, in addition to conventional loop iteration-level parallel processing. Further, we parallelized engine control programs, which have poor loop parallelism. At this time, branch duplication and task-fusion were employed to make task granularity coarser and enable static scheduling to cope with basic block fine granularity avoiding dynamic scheduling overhead. While they can be also efficiently used for Ladder programs, finer task granularity in a target program must be overcome.

This paper proposes an acceleration technique for Ladder programs by OSCAR automatic parallelizing compiler. It first translates a source Ladder program into a C program. Then, the compiler parallelizes it. At the parallelization, the compiler exploits

coarse grain task parallelism from a translated C program. The compiler also combines coarse grain tasks considering available parallelism to mitigate synchronization overhead, in addition to previously proposed branch-duplication and task-fusion for hiding if-statements. Finally, the parallelized program is statically scheduled on processor cores.

This paper includes the following contributions:

- We developed a Ladder-to-C translator. It enables the OSCAR compiler to parallelize Ladder programs.
- We propose a task-fusion technique to mitigate synchronization overhead from fine task granularity in a Ladder program.
- We conducted an experimental evaluation using industry-provided programs. It reveals the proposed technique can reduce 17% of execution clock cycles.

The rest of the paper is organized as follows. Section 2 introduces the related works. Section 3 describes Ladder language. Section 4 introduces the proposed Ladder transformation method in this paper. Section 5 provides the overview of OSCAR automatically parallelization compiler. Section 6 shows the evaluation results of our proposed methods on the application from the industry. Section 7 concludes the paper.

2 Related Works

Vasu proposed a parallelization technique for Ladder programs called Soft-PLC. [8]. It converts a Ladder program into an intermediate representation, performs dependency analysis on it for each circuit block using a Python program, and achieves speed-up through parallelization by using Python's multi-process execution model.

Regarding parallelization of control programs, several studies on model-based design such as MATLAB/Simulink have been studied. Zhong achieved parallelization by using parallelism between blocks in a model [10]. Umeda realized speed-up by exploiting parallelism within blocks in addition to the parallelism between blocks in a model [7]. Similar to the model-based development, a Ladder program can be represented as a block diagram. However, its program structure is difficult to grasp resulting in the difficulty of parallelization. To overcome this problem, this paper proposes a Ladder program-to-C translator. OSCAR compiler takes the result C code by the translator, exploits parallelization, and generates a parallelized C code by inserting OpenMP or OSCAR-API directives. In addition, the compiler generates a macro task graph that represents the data and control dependencies of the program, making it possible to easily check the program structure.

3 Ladder Language

A Ladder program is a model of sequence control [9], which was conventionally performed by relays and switches. There are two types of representations of a Ladder program: Ladder diagrams, which directory represent a control circuit as a block diagram, and instruction lists (ILs), which represent a Ladder diagram in a text format. A

developer usually develops a program in a Ladder diagram on a development tool, and it can output ILs from the diagram format.

A Ladder diagram consists of Ladder rungs and Ladder instructions. Ladder instructions are connected with both ends of a Ladder rung. A circuit that starts with an instruction connected to the left Ladder rung and ends with an instruction connected to the right Ladder rung is called a circuit block. A Ladder diagram is composed by connecting these circuit blocks. A circuit block represents an instruction and its execution condition. A Ladder instruction referring to a memory value is called a device. Typical devices include input X and output Y with logic values, internal relay M that holds bit information inside the PLC, device K that holds an immediate value, word device D that handles 16-bit word data, timer device T that measures time, and counter device C that counts numbers. A device is represented by a pair of device symbol and number, which indicates a location of the device to be accessed. For example, X3 uses the value of the third element of input device X.

In a conditional part of a circuit block, (1) “Open contact” and (2) “Close contact” in Figure 1 are used. An open contact in this figure is turned on when X0 is 1, and a close contact is turned on when X0 is 0. An execution part of a circuit block uses (3) “OUT instructions” and so on. An OUT instruction is a special instruction that is always executed and holds 1 in M0 when the condition is satisfied and holds 0 when the condition is not satisfied. Other instructions are executed when a condition is satisfied, and there are various instructions such as data transfer instructions and four arithmetic operations.

Symbols used in Ladder diagrams are expressed in IL language such as LD, AND, OUT, +, and so on. A conditional part can have LD and LDI, which handle open and close contacts respectively as contact start instructions, and AND, OR, ANI, and ORI, which combine contacts. An execution section uses IL instructions, including an OUT instruction shown in (3) of Figure 1, a “+” instruction for addition, a “-” instruction for subtraction, and a MOV instruction for data transfer.

A Ladder program is executed from left to right and top to bottom. Then, when an END instruction is executed, it is executed again from the top to the bottom. Figure 2 shows an example of a Ladder diagram and Figure 3 shows a corresponding Ladder ILs, respectively. The execution flow of this program is as follows:

1. If X0 is closed, M0 sets as 1 (close), else M0 sets as 0 (open).
2. If X2 and M0 are closed, add 2 to D2.
3. If X2 and M1 are closed, subtract 2 from D2.
4. Go back to step 1 unconditionally.

Execution time from the first instruction to the END instruction is called scan time, and there is a demand for faster scan time.

4 Ladder Program Transformation

This section explains the proposed Ladder-to-C translator. Considering the fact that a Ladder program consists of conditional parts and execution parts, the translator translates a circuit block in a source Ladder program into an if-statement in an output C program.

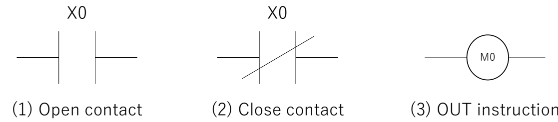

Fig. 1. Symbols used in Ladder diagrams

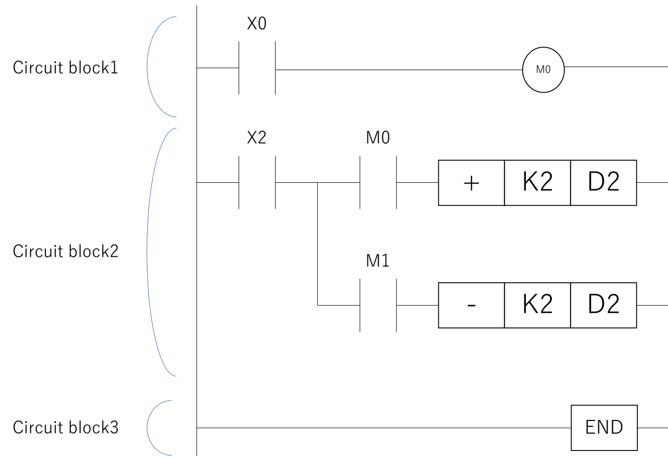

Fig. 2. Example of a Ladder diagram. A Ladder diagram consists of Circuit blocks. They start with a part connected to a power rail on the left side. Circuit block1 includes an open contact X0 and OUT instruction for M0. Circuit block2 includes 3 contacts and 2 instructions. Circuit block3 includes only END instruction.

4.1 Translation of Instructions around Contacts

An open contact by an LD instruction or a close contact by an LDI instruction is translated into a single if clause to realize its conditional behaviors explained in Section 3. In addition, a Ladder program has MPS instructions to push contact operation results onto the stack, MRD instructions to load from the stack, and MPP instructions to pop from the stack. Our translator handles these three instructions as follows.

- MPS: Start an if clause using the result of the contact operation up to the immediately preceding point, and execute the subsequent instructions inside this if clause.
- MRD: Output an if clause using the results of the contact operations up to the previous point.
- MPP: Output an if clause using the results of the contact operations up to the previous point. After the output of the immediately preceding result section is completed, close one if clause.

4.2 Device Handling in Ladder Translator

The proposed translator translates devices representing data in a source Ladder program into arrays in a generated C program. At this time, each device number explained Section

```
LD      X0
OUT     M0
LD      X2
MPS
AND     M0
 +      K2 D2
MPP
AND     M1
 -      K2 D2
END
```

Fig. 3. Example of a Ladder instruction list. The left and right sides show instructions and arguments, respectively.

3 corresponds to each array element in the generated array in the C program. By doing so, the compiler can analyze data dependency among Ladder operations accessing the devices, resulting in exploitation of the parallelism from the Ladder program.

4.3 Translation Example

Figure 4 shows the example of translation by the Ladder-to-C translator. It translates LD instructions into if clauses. As described in section 3, an OUT instruction is executed unconditionally. Thus, as shown in the first five lines on the right side of Figure 4, the result of “LD X0” is once stored in a temporal variable, then an assign statement corresponding “OUT M0” stores the value in the temporal variable into “M[0].” MPS instruction starts an if clause using the X2 from the previous LD instruction. AND instruction after the MPS instruction also starts if clause to express the execution condition for the add instruction in Figure 3. After translating the add instruction, the translator closes the if clause started by the AND instruction placed before the add instruction. MPP instruction and AND instruction start the if clause using the device M1. After translating the subtraction instruction, the translator closes both if clauses opened by the AND and MPS instructions. As mentioned in Section 4.2, all devices in the Ladder are translated into arrays. Note that the device K holds an immediate value, therefore it is translated into an integer value that it holds, instead of an array element.

5 OSCAR Automatic Parallelizing Compiler

5.1 Macro Task Level Parallelization

One of the main features of the OSCAR automatic parallelizing compiler is exploitation of Macro Task (MT) level parallelism in addition to conventional loop-iteration level parallelism and statement level near-fine grain parallelism. The compiler defines basic blocks, loops, and function calls in a source program as macro tasks. For Ladder programs, almost all macro tasks are basic blocks.

In macro task level parallel processing, or coarse grain task parallel processing, the compiler divides a source program into macro tasks. Then, the compiler analyses

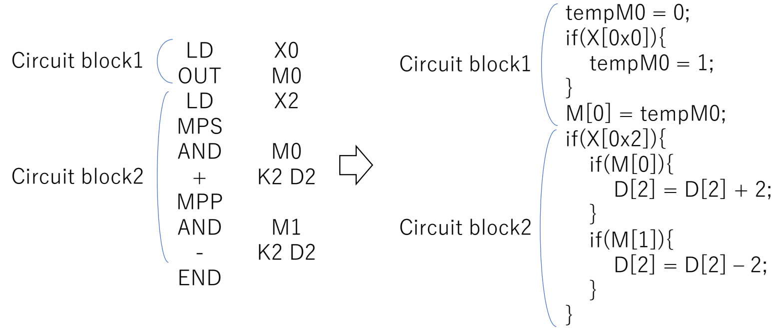

Fig. 4. Translation example of a Ladder diagram shown in Figure 2. One circuit block translated into one if clause.

control flow and data dependencies among them. The analysis result is represented as a macro flow graph (MFG). Figure 5 shows an example of MFG. In the figure, a node represents a macro task. A small circle at a bottom of an MT represents a conditional branch. A solid edge represents a data dependence and a dotted edge represents a control dependence, respectively. Next, the compiler performs the earliest executable condition (EEC) analysis from an MFG. For each macro task, an earliest executable condition represents when the macro task can start its execution the earliest considering its data dependence and control dependence. The compiler generates a macro task graph (MTG) as a result of the earliest executable condition analysis, which naturally represents the parallelism among macro tasks.

Figure 6 shows an example of MTG. A solid edge represents data dependence and a dotted edge represents a control dependence, respectively. A solid arc in front of a macro task is an AND arc, representing the macro task can start when all data and control dependencies bound by it are satisfied. Similarly, a dotted arc is an OR arc, which represents the macro task can start when one of data and control dependencies bound by it is satisfied.

After generating a macro task graph, macro tasks are assigned to cores in a target multicore for parallel execution. At this time, the compiler chooses dynamic scheduling or static scheduling. Dynamic scheduling determines the allocation at the program execution time, while static scheduling determines it at the compile time. Dynamic scheduling can deal with conditional branches and task execution cost fluctuation appropriately, while it introduces runtime scheduling overhead. On the other hand, static scheduling has no scheduling overhead at runtime because it can schedule in advance when an MTG has no conditional branch [6].

5.2 Basic Block Decomposition

If a macro task is a basic block and it can be decomposed into independent groups of statements, the compiler can decompose it into multiple macro tasks. To do so, the compiler also builds a task graph representing data dependencies among statements in a basic block. According to the built task graph, the compiler can detect the independent

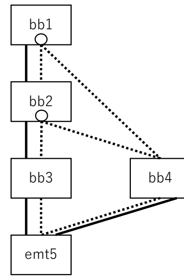

Fig. 5. MFG sample. A small circle at a bottom of an MT represents a conditional branch. A solid edge represents a data dependence and a dotted edge represents a control dependence, respectively.

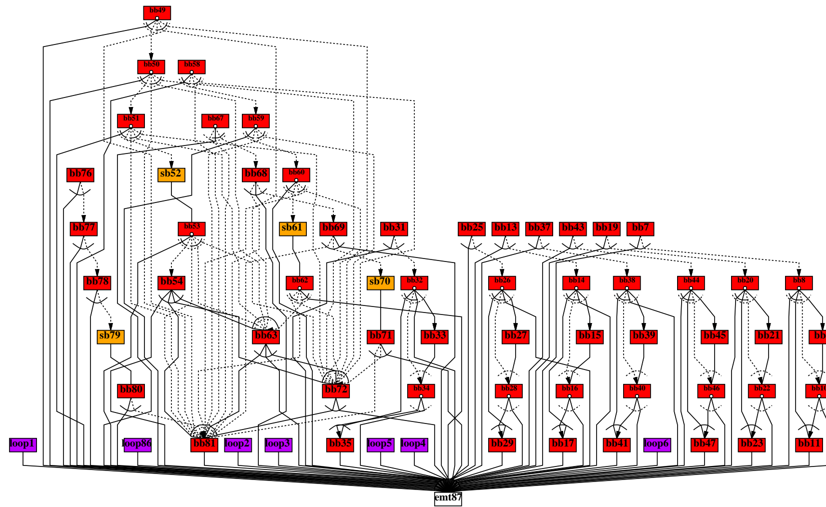

Fig. 6. MTG sample. It is a graph that adds EEC analysis results to an MFG.

groups of statements. Thus, the compiler can exploit more parallelism. This basic block decomposition makes finer macro tasks resulting in relatively larger synchronization and data transfer overhead among macro tasks. To mitigate it, the compiler tries to fuse multiple macro tasks as explained in the following subsections.

5.3 Avoiding Dynamic Scheduling Overhead

A plant control program, like a Ladder program, has many operations, which require conditional branches since their behavior is determined by sensor inputs. Thus, dynamic scheduling is required to deal with this dynamic behavior of a program. However, the execution cost of each macro task in a plant control program tends to be small, resulting in the impracticality of dynamic scheduling [7].

To avoid dynamic scheduling, the compiler fuses a macro task containing a conditional branch and macro tasks that have control dependent on the conditional branch

macro task so that the fused macro task can hide conditional branches inside it. Therefore, the compiler can employ static scheduling for an MTG after this macro task fusion. It also enlarges the execution cost of a macro task.

Figure 7 shows the result of applying this technique to Figure 6. All control dependencies are hidden in the macro tasks.

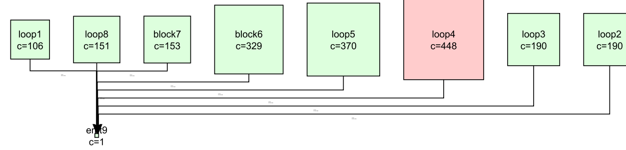

Fig. 7. MTG sample(control flow dependence free). All conditional branches are hidden in the macro tasks. The edges represent the data dependencies.

5.4 Branch Duplication

Although the macro task fusion explained in Section 5.3 can avoid dynamic scheduling by hiding conditional branches, this may result in the loss of parallelism within a conditional branch. For instance, when a then-part consists of two independent macro tasks, the macro task fusion technique spoils it since they are fused into the same macro task. To exploit the original parallelism in this case, we proposed branch duplication technique [2]. For this example, it duplicates the conditional branch for those independent macro tasks in the then-part, and each pair of the duplicated conditional branch and a macro task in the then-part are fused in one macro task. Thus, both of avoiding dynamic scheduling and exploitation of original parallelism can be realized.

5.5 Macro Task Fusion for Ladder Programs

As mentioned in section 5.3, Ladder programs consists of small macro tasks. In addition, they frequently contain instruction sequences such that an if-clause including variable accesses follows its initialization statement. The statements in each of them are data-dependent and worth task fusion. To reduce a synchronization overhead and a data transfer overhead, we implemented another macro task fusion technique. It fuses macro tasks for the following four cases:

- For two macro tasks MT X and MT Y, if MT X has only successor MT Y, X and Y are fused.
- For two macro tasks MT X and MT Y, if MT X has only predecessor MT Y, X and Y are fused.
- If macro tasks have no predecessor macro tasks and they also have common successor macro tasks, they are fused into a single macro task. For instance, in Figure 8, MT1 and MT2 are fused into MT1-2. Similarly, MT3 and MT4 are fused into

MT3-4. This kind of MTs frequently appear in Ladder programs to initialize device values.
- If macro tasks have no successor macro tasks other than the end macro task (EMT), which is the last macro task of an MTG, and they also have common predecessor macro tasks, they are fused into a single macro task. For instance, in Figure 9, MT3 and MT4 are fused int MT3-4. Similarly, MT5 and MT6 are fused into MT5-6. This kind of macro tasks correspond to setting final result to external devices, and they are independent from other macro tasks. They also frequently appear in Ladder programs.

Note that, this fusion technique is only employable to macro tasks whose execution cost are less than synchronization and data transfer costs to avoid spoiling available parallelism.

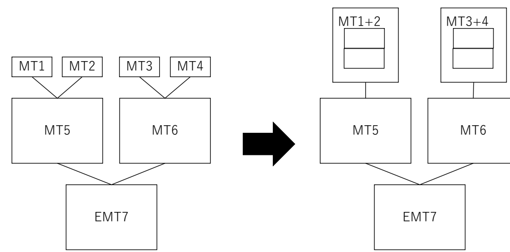

Fig. 8. Macro task fusion for MTs with no predecessors. The upper four MTs are fused, two by two.

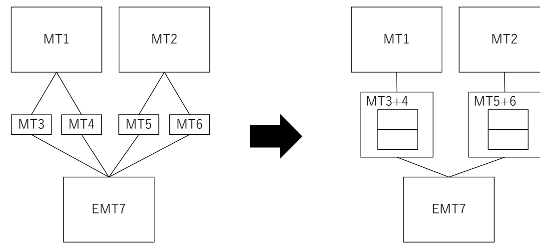

Fig. 9. Macro task fusion for MTs with no successors. The middle four MTs are fused, two by two.

6 Evaluation

6.1 Evaluation Environment

We use Xilinx ZCU102 board with Cortex-A53 driven at 300MHz (4 cores) and 4GB memory [1] for the evaluation. Ubuntu 20.04.2 LTS is installed on it.

6.2 Evaluation Programs

We evaluated three proprietary factory automation small test programs. Table 1 shows their summary as the numbers of lines of and the execution clock cycles. We use them since they appropriately represents real Ladder programs for factory automation usage and there are no publicly available such Ladder programs. They are labeled from “Program1” to “Program3”. Note that they have many basic blocks and a few loop. Thus, ordinary product parallelizing compilers cannot exploit parallelism from the programs composed of a set of basic blocks.

6.3 Evaluation Results

Table 2 shows the average task cost obtained before and after task fusion for each evaluated program. Here, “cost” is the estimated clock cycles for a virtual target multicore modeled in the compiler. Figure 11 shows the Program2’s MTG, and Figure 13 (a) shows that for Program3, respectively. Similarly, Figures 10, 12, and 13 (b) show the MTGs for Program1, 2 and 3, respectively. The task fusion technique described in Section 5.5 is employed for them. As described in Section 5.3, a coarse grain parallelization for small macro tasks is ineffective due to the data transfer and synchronization overhead. According to Table 2, our task fusion technique makes task granularity about twice as large for Program1, about 6 times for Program2, and about 10 times for Program3, respectively. Thus, these results indicate a relative reduction in parallelization overhead.

Table 3 shows the summary of the parallelism exploited from the evaluated programs. Here, “parallelism” is calculated as “total task cost” divided by “critical path length” by the compiler [5]. In this table, the parallelism of Program1 and Program2 is still greater than 2 after the newly proposed task fusion. Hence, the newly proposed task fusion technique described in section 5.5 can maintain sufficient parallelism for two cores for Program1 and Program2. Although parallelism of Program3 is less than 2 after employing our proposal, task granularity increased 10 times, suggesting that the actual execution performance is better.

Figure 14 shows the clock measurement results for Program2 on Xilinx ZCU102. According to this figure, the Ladder-to-C translator and GCC’s execution clock on one core were 1614. The execution clock on two cores was increased to 3633, namely 2.25 times slower than sequential execution, by the Ladder-to-C translator and the OSCAR compiler without task fusion considering data transfer and synchronization overhead due to a synchronization overhead and a data transfer overhead. The execution clocks on two cores were reduced to 1335 by the Ladder-to-C translator and the OSCAR compiler, which implements the macro task fusion method described in section 5.5. In summary, our current task fusion allows us to speed up 1.2 times on an actual Arm SMP multicore for the first time in the past 50 years.

6.4 Comparison with Soft-PLC

As mentioned in Section 2, Soft-PLC also realized Ladder program parallelization [8]. Different from our parallelization technique, it did not handle indirect device accesses by an index register. Our translator translates a device into an array and it can naturally

handle indirect accesses. In addition, it was a kind of a simulator that executes Ladder programs though it realized parallel execution. On the other hand, our technique can generate a parallelized native code that can be directly executed on an Arm multicore. Finally, we conducted the experimental evaluation with industry-provided Ladder programs as shown in this section, while the Soft-PLC was evaluated with randomly generated Ladder programs having around several dozen steps.

Table 1. Summary of the evaluated programs

	Number of steps	Execution clock cycles of translated Ladder programs on ZCU102
Program1	196	269
Program2	1766	1614
Program3	522	383

Table 2. Summary of average task cost estimated in the compiler

	Average task cost without macro task fusion	Average task cost with macro task fusion
Program1	101.95	276.86
Program2	236.30	1,394.92
Program3	36.95	363.71

Table 3. Summary of the parallelism estimated in the compiler

	Parallelism without macro task fusion	Parallelism with macro task fusion
Program1	4.3	4.3
Program2	2.5	2.0
Program3	2.0	1.4

7 Conclusion

This paper has proposed a parallelizing compilation method for "Ladder" programs used for Factory Automation (FA) for over 50 years. A ladder program is translated into C by a newly developed translator in the proposed method. Next, the OSCAR compiler parallelizes the generated C program for any shared memory multicores with or without a coherent cache. The generated C program consists of small basic blocks with control

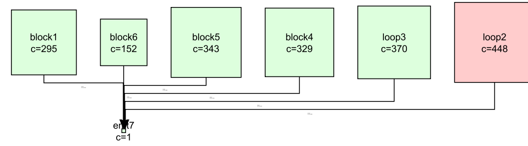

Fig. 10. MTG for Program1 (Newly proposed task fusion described in Section 5.5 was employed.)

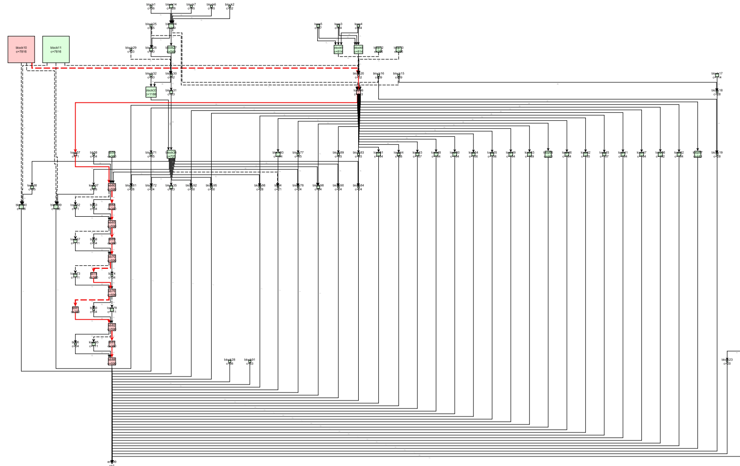

Fig. 11. MTG for Program2 (Task fusion described in Section 5.3 was employed.)

dependencies. OSCAR Compiler generates a macro task graph composed of basic blocks as macro tasks. It fuses the micro tasks into other micro tasks having only data dependencies among them by task fusion technique developed for "automobile engine control program" parallelization. Next, the OSCAR compiler fuses small micro tasks into coarser ones, considering task execution time, data transfer, and synchronization overhead. By the proposed compilation scheme, this paper succeeded in parallelizing the Ladder programs on a real multicore processor, although there has not existed automatic parallelization of the ladder circuit for over 50 years. In the evaluation, since the actual ladder programs directly represent existing control systems are difficult to use, we we used a few small proprietary test programs to evaluate compilation analysis and an executable program with input and output data. This paper conducted an experimental evaluation on an Arm two-core multicore. The execution time on two cores without the proposed task fusion considering data transfer and synchronization overhead, was 2.25 times slower than sequential execution. However, the proposed task fusion allows us to speed up 1.2 times on an actual Arm SMP multicore for the first time in the past 50 years. The proposed Ladder-to-C translation and the macro task fusion to reduce data transfer and synchronization overhead has shown for the first time that parallel processing of the real-time sequence control computation on an SMP multicore is possible. Improvement of the macro-task fusion method will allow us more speedups. Also, this method can be

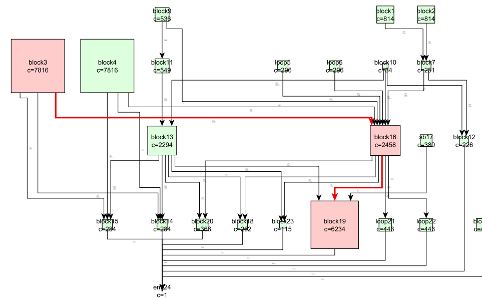

Fig. 12. MTG for Program2 (Newly proposed task fusion described in Section 5.5 was employed.)

easily applied to embedded multicores using distributed shared memory like Renesas and Infineon for automobiles, like in our previous paper [7].

References

1. Zynq ultrascale+ mpsoc zcu102 evaluation kit. `https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html`
2. Kanehagi, Y., Umeda, D., Hayashi, A., Kimura, K., Kasahara, H.: Parallelization of automotive engine control software on embedded multi-core processor using oscar compiler. In: 2013 IEEE COOL Chips XVI. pp. 1–3. IEEE (2013)
3. Kasahara, H., Honda, H., Mogi, A., Ogura, A., Fujiwara, K., Narita, S.: A multi-grain parallelizing compilation scheme for oscar (optimzally scheduled advanced multiprocessor). In: Proc. 4th Intl. Workshop on LCPC. pp. 283–297 (August 1991)
4. Kasahara, H., Obata, M., Ishizaka, K.: Automatic coarse grain task parallel processing on smp using openmp. In: Languages and Compilers for Parallel Computing. pp. 189–207. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
5. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierarchical parallelism control for multigrain parallel processing. In: Languages and Compilers for Parallel Computing. pp. 31–44. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
6. Oki, Y., Mikami, H., Nishida, H., Umeda, D., Kimura, K., Kasahara, H.: Performance of static and dynamic task scheduling for real-time engine control system on embedded multicore processor. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 1–14. Springer (2019)
7. Umeda, D., Suzuki, T., Mikami, H., Kimura, K., Kasahara, H.: Multigrain parallelization for model-based design applications using the oscar compiler. In: Languages and Compilers for Parallel Computing. pp. 125–139. Springer (2015)
8. Vasu, P., Chouhan, H., Naik, N.: Design and implementation of optimal soft-programmable logic controller on multicore processor. In: 2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS). pp. 1–4. IEEE (2017)
9. W.Bolton: Programable Logic Controllers Sixth Edition. Newnes (2015)
10. Zhong, Z., Edahiro, M.: Model-based parallelizer for embedded control systems on single-isa heterogeneous multicore processors. In: 2018 International SoC Design Conference (ISOCC). pp. 117–118. IEEE (2018)

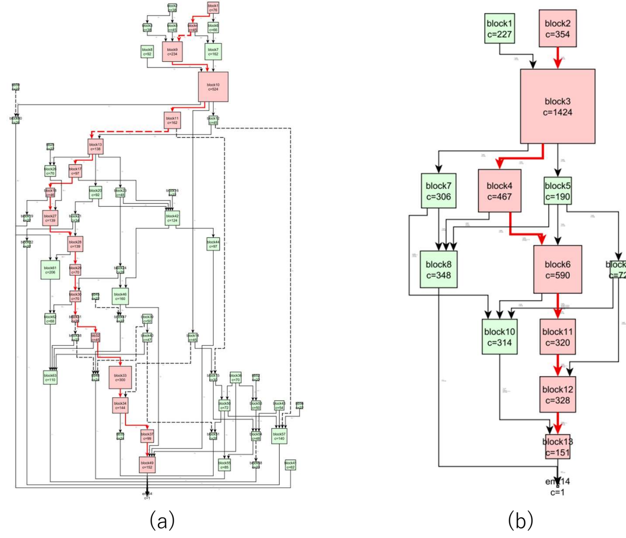

Fig. 13. MTGs for Program3. (a) was employed the task fusion technique described in Section 5.3. (b) was employed the newly proposed task fusion technique described in Section 5.5.

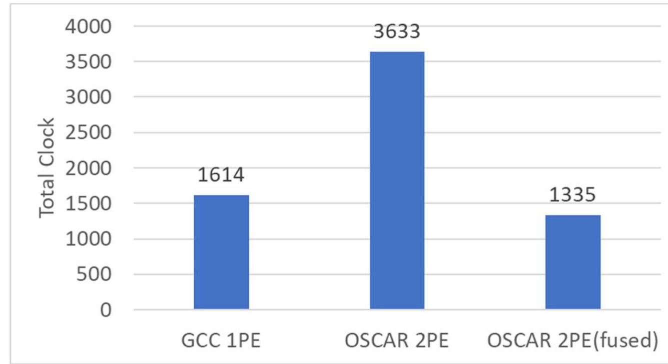

Fig. 14. Clock measurement results on the ZCU102. The left bar is the result obtained by compiling the output of our Ladder-to-C translator with GCC. The center bar is the result with the task fusion technique described in Section 5.3. The right bar is the result with the newly proposed task fusion technique described in Section 5.5.