Fast and Highly Optimizing Separate Compilation for Automatic Parallelization

Tohma Kawasumi
Waseda University
Tokyo, Japan
tohma@kasahara.cs.waseda.ac.jp

Ryota Tamura
Waseda University
Tokyo, Japan
r_tamura@kasahara.cs.waseda.ac.jp

Yuya Asada
Waseda University
Tokyo, Japan
yu_asd@kasahara.cs.waseda.ac.jp

Jixin Han
Waseda University
Tokyo, Japan
kalfazed@kasahara.cs.waseda.ac.jp

Hiroki Mikami
Waseda University
Tokyo, Japan
hiroki@kasahara.cs.waseda.ac.jp

Keiji Kimura
Waseda University
Tokyo, Japan
keiji@waseda.jp

Hironori Kasahara
Waseda University
Tokyo, Japan
kasahara@waseda.jp

***Abstract*—Automatic parallelization by a compiler is a promising approach for fully utilizing a multicore processor. Without compiler support, a programmer must simultaneously take into account parallelism in a program and memory hierarchy utilization. However, the possibility of parallelization and optimization across multiple compilation units is limited due to the lack of interprocedural analysis information at the compile time. This is a serious challenge surrounding parallelizing practical programs because they usually consist of multiple compilation units and employ separate compilation to ensure program maintainability and reduce the recompilation time. In this paper, for automatic parallelization by a compiler, we propose a separate compilation method that enables parallelization across multiple compilation units and minimizes recompilation time by providing information about the analysis along with an object file for each compilation unit at the compile time. We also propose an automatically parallelizing compilation flow with analysis information. The experimental evaluation using large size real control system programs from industry shows the proposed technique can obtain 29% better performance than the separate compilation without the proposed method, and reduce compilation time by up to 90% with only 1% of performance loss, compared with the compilation for the fully unified source code into a single compilation unit.**

***Index Terms*—parallelizing compiler, program analysis, link time optimization**

I. Introduction

Parallel programming APIs and parallelizing compilers are widely used because multicore and manycore systems are ubiquitous. Programs must be parallelized to fully exploit the performance of those systems [3], [4], [8]–[11]. Investigating only parallelism from a program is not enough to obtain higher performance from them. A programmer also needs to carefully consider utilizing memory hierarchy on the target system. Furthermore, reducing power consumption while keeping higher performance is becoming important. Developing a parallelized program taking into account all of them is a difficult task for an ordinary programmer even with a good parallel API. However, automatic parallelization by a compiler can overcome this challenge [3], [7].

While employing automatic parallelizing compilers is a promising approach, it comes with several considerations during the development of a large practical program. A compiler must employ pointer analysis and data-flow analysis as widely as possible to exploit parallelism from a program; without analysis result, it must stop parallelization in a part of the program where enough analysis information cannot be obtained, to avoid faulty parallelization. Thus, a compilation unit should be satisfactory large such that the program analysis can exploit existing parallelism in a program. On the other hand, the compiler may also employ code restructuring techniques, such as inlining, for a program multiple times for parallelization; these are followed by costly program analysis passes to update the analysis information. Those characteristics of program analysis and restructuring increase both compilation time and the size of the compilation unit [16], [17].

The expensive parallelizing compilation cost becomes significant especially for large practical programs. They usually consist of multiple compilation units to keep program maintainability and reduce recompilation time. If all the compilation units are unified into a single file to apply program analysis across multiple original compilation units, it introduces large recompilation time, even if only a small part of the program is modified.

Link Time Optimization (LTO) has been used to employ interprocedural analysis and restructuring techniques across multiple compilation units [1], [2], [5], [12], [13]. They provide compile time information, such as the intermediate representation (IR) and analysis information, of each compilation unit along with an object file. Then, they are integrated at the link time to employ compiler optimization techniques for the entirety of a program.

Though the LTO is also valuable for automatic parallelization, there are two limitations arising from simply employing previously proposed LTO techniques: the large recompilation time and the compilation order among the compilation units.

The LTO basically provides the IR and integrating all of them at the link time [1], [5], [12]. This actually enables compiler optimization techniques across multiple compilation units. However, this cannot reduce recompilation time because the ordinary compilation passes are employed for the entirety of the integrated program.

To reduce the recompilation time, ThinLTO was proposed [2], [13]. It generates analysis information for each compilation unit and integrates them at the link time. Thus, it can reduce recompilation time because a compiler does not have to employ program analysis in a compilation unit if it was not modified. However, for automatic parallelization, the compiler should distinguish the types of compilation units. The program to be parallelized usually consists of one, or at most a few parallelizable parts that occupy much of program execution time, and many functions called by parallelized parts, for which a compiler does not have to employ parallelization. In addition, if a function is worth employing interprocedural restructurings for, it should be combined with a compilation unit that includes parallelized parts to employ those restructurings. Those compilation processes and the order of compilation units must be defined in the flow for automatic parallelization.

In this paper, we propose a separate compilation flow for automatic parallelization. It firstly categorizes the compilation units into two groups: the parallelizing group and the sequential group. The compiler then employs program analysis passes for compilation units of the sequential group, and generates program analysis information files. Then, the compiler utilizes the information files to parallelize the compilation units of the parallelizing group. Thus, it can ensure both of the reduction of recompilation time and the capability of parallelization.

This paper includes the following contributions:

- The contents of the analysis information that enable automatic parallelization across multiple compilation units are specified.
- A separate compilation flow for automatic parallelization is proposed. It categorizes multiple compilation units into two groups to employ interprocedural parallelization across original compilation units. For this grouping, the manual approach and automatic approach are prepared. It also specifies the order of compilation, taking the types of compilation units into consideration.
- The experimental evaluation using large size real control system programs from industry shows that the proposed separate compilation flow can obtain 29% better performance than the separate compilation without analysis information, and can reduce compilation time by up to 90%, with only 1% loss of performance, compared with the compilation on a single file unifying all compilation units when the automatic grouping is employed.

The rest of this paper is organized as follows: Section II reviews the overview of the LTO and discusses the challenges of employing it for automatic parallelization. Section III provides the overview of the automatic parallelizing compiler to clarify the requirement for analysis information for automatic parallelization. Section IV introduces the proposed method in this paper. Section V shows the experimental evaluation results. Finally, Section VI concludes this paper.

II. Related Works: Link Time Optimization (LTO)

Link Time Optimization (LTO) has been used to overcome the limitation of separate compilation that cannot employ interprocedural optimization in some part of a program where a compiler cannot use the information of other compilation units. The basic idea of the LTO is utilizing some program information of other compilation units at compile time. It consists of two steps. In the first step, the compiler performs compilation for each compilation unit as usual. In the second step, the compiler integrates the information of compilation units and employ interprocedural optimization passes.

In terms of the types of compilation information to be integrated, there are mainly two kinds of LTO techniques. Here, refer to them as (1) IR-based LTO and (2) Analysis information-based LTO. Because the proposed method in this paper can be seen as an LTO technique in terms of the above idea, we briefly review the two kinds of LTO in the following subsections.

A. Intermediate Representation (IR) based LTO

For the IR-based LTO, all available IR from all the compilation units are integrated to employ interprocedural optimization passes. Figure 1 shows the overview of the compile flow for IR-based LTO. The IR derived from each compilation unit is embedded into its object file at the first compilation step as a large object file. At the second compilation step, a linker integrates all the IR of those compilation units and employs interprocedural optimization passes. In this step, the linker traverses the entire source program by using the IR in all the available large object files. Finally, the optimized executable binary is generated.

This approach can perform optimization passes on the source program as if it consists of a single file. Though this approach can fully employ interprocedural optimization passes, it cannot reduce the compilation time. This is because the compiler must traverse the entire program at the link time.

This approach has been used by GCC, LLVM, and other compilers [1], [5], [6], [12].

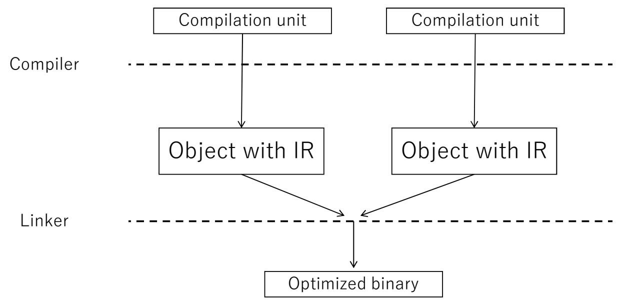

Fig. 1. Overview of the compile flow for IR-based LTO. The compiler generates the object with IR of each compilation unit.

B. Analysis Information-based LTO

For the analysis information-based LTO, the analysis information (or summary information) from all the compilation units is integrated, instead of the IR incorporating all their program structure [2], [13]. Figure 2 shows the overview of the compile flow for the analysis information-based LTO. A compiler generates analysis information from each compilation unit as well as its object file in the first step. The generated information may be embedded inside the object file. Then, a linker integrates all the available analysis information to perform interprocedural program analysis in the second step. From the analysis result, the compiler optimizer can detect the program location where interprocedural optimization passes is effective, and can deploy it with the corresponding IR.

This approach can reduce the compilation and the recompilation time because the interprocedural program analysis can be performed on the analysis information of the target program, instead of on relatively larger IR from all compilation units.

While this approach is scalable against the size of a source program, it is difficult to simply employ it for the automatic parallelizing compiler. Program parallelization must consider both exploiting parallelism and reducing parallel execution overhead. When a part of a program has little parallelism, the compiler should avoid employing expensive analysis. Therefore, a parallelizing compiler should categorize compilation units into two groups, such as a sequential group and a parallelizing group, and employ the appropriate compilation for them. On the other hand, existing analysis information-based LTO techniques deal with all the compilation units evenly. The main uniqueness of our technique is that it builds a separation compilation flow based on this insight.

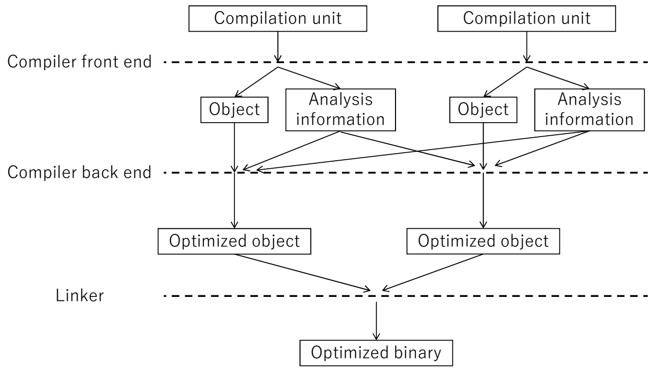

Fig. 2. Overview of the compile flow for the analysis information-based LTO. The compiler front end generates the object and analysis information file of each compilation unit. Then, compiler back end optimizes the object of the compilation unit by using the analysis information of other compilation units.

III. OSCAR Automatic Parallelizing Compiler

In this section, to clarify the information that supports separate compilation for automatic parallelization by a compiler, we first review the overview of the OSCAR automatic parallelizing compiler as an example of a parallelizing compiler; the required analysis information is discussed based on it. Though the OSCAR compiler is reviewed here, the required analysis information can be used for other parallelizing compilers.

A. Overview of OSCAR Compiler

The OSCAR automatic parallelizing compiler takes an ordinary sequential C or Fortran program and generates a parallelized code annotated by OpenMP or OSCAR-API directives [3], [7]. It employs multi-grain parallel processing that can exploit multiple grains of parallelism hierarchically.

The compiler consists of the front-end, the middle-end, and the back-end. The front-end takes a source program and translates it into a form of intermediate representation, OSCAR-IR. The middle-end performs parallelization on the IR along with multiple program analysis and restructuring passes. Finally, the back-end generates the parallelized C or Fortran code, then a target platform compiler, such as GCC, generates an object file from it.

The OSCAR compiler first decomposes a source program into three kinds of blocks: basic block (BB), repetition block representing an outer most loop (RB), and subroutine block representing a function call statement (SB). Each block is called a macro-task (MT).

Then, the compiler employs control flow analysis, pointer analysis, data-flow analysis, and data-dependence analysis like the other compilers. It also employs several restructuring techniques to exploit parallelism as much as possible.

The results of the control flow analysis and data dependence analysis are represented as a macro-flow graph (MFG), where each node represents an MT. Then, the compiler employs the earliest executable condition analysis that can exploit the earliest MT starting condition from the control dependence and data dependence of each MT. The analysis results are represented as a macro-task graph (MTG). Thus, the compiler can exploit parallelism from the MTs, namely coarse grain task parallelism.

If the compiler can detect coarse grain task parallelism inside the MT of the RB or SB, the compiler hierarchically exploits it. If the RB is a parallelizable loop (do-all/for-all), the compiler employs loop iteration level parallelism for it, similar to ordinary parallelizing compilers.

Finally, the compiler decides the scheduling policy for each MTG and generates the parallelized code. If an MTG has conditional branches in it, the MTs in it are dynamically scheduled to processor cores at the runtime. The compiler generates the runtime scheduling code for that MTG as well as the code for each MT as the parallelization result. Otherwise, the static scheduling is employed for it to minimize the runtime overhead caused by the scheduling code.

As described in the above, the OSCAR compiler can hierarchically exploit parallelism from a program. While this means that the compiler can fully exploit the parallelism from the program, it also requires that the number of processor cores for each part of the program must be appropriately decided depending of the amount of the parallelism there. This is because the parallel execution for the part with little parallelism suffers from the large synchronization overhead. For this purpose, the OSCAR compiler calculates the parameter $Para$ for each

MTG [15]. $Para$ is calculated by the following equation:

$$Para = Cost/CriticalPathLength \tag{1}$$

Here, $Cost$ is the sum of execution costs of all the MTs in the MTG, and $CriticalPathLength$ is the critical path length of the MTG, respectively. In short, $Para$ indicates the magnitude of the parallelism in the MTG. For instance, if $Para$ is less than 2, the compiler stops to parallelize that MTG anymore.

The compiler estimates the execution cost of an MT to calculate $Cost$ in the equation 1 by accumulating all operation costs in it. The profile result can be also used to obtain more accurate $Cost$.

Throughout the compilation flow described above, the program structure and MTGs as well as other information related with parallel processing are maintained by OSCAR-IR.

B. Analysis Information Supporting Separate Compilation for Automatic Parallelization

Now, the required analysis information that enables parallelization over multiple compilation units is discussed according to the above compilation flow.

Needless to say, the information of the defined and used variables of a function in another compilation unit must be present because the parallelism analysis relies on the data dependence among tasks for task parallel processing, or iterations in a loop for loop iteration-level parallel processing. Access range information is also useful for arrays. For pointer variables, the defined and used information of an object pointed to by a pointer variable should be provided.

In addition, there are several considerations if a function in another compilation unit modifies the function scope or the compilation unit scope static variables. When a function modifies a function scope static variable, its multiple function call must not be executed in parallel (1). On the other hand, when multiple functions have data dependence on each other through having the same compilation unit scope static variable, they too must not be executed in parallel (2). Furthermore, when a function modifies a machine's global state, for instance, by accessing some I/O peripherals, it must not be executed with other tasks (3). A parallelizing compiler must consider those situations. However, the variables causing those dependencies are not visible from other compilation units. Therefore, the analysis information should include the attributes of each function, such as "having a state" for the case (1), "dependent on other functions in same compilation unit" for the case (2), and "sequential" for the case (3).

In terms of the task scheduling phase, cost information is important. Therefore, the analysis information should include it for each function.

IV. Separate Compilation for Automatic Parallelization

In this section, we explain the proposed separate compilation flow. This is similar to the analysis information-based LTO described in Section II-B. Before the separate compilation, the compilation units of a source program are divided into two groups. Then, they are appropriately ordered and employed for sequential or parallelizing compilation for the flow. The recompilation with the proposed flow is also described here.

A. Code Separation

Before the compilation, the compilation units of a source program are divided into two groups, the parallelizing group and the sequential group, to decide the order of the compilation among them and to perform appropriate analysis and optimization passes on them.

- **Parallelizing Group**
  A compilation unit including program parts to be parallelized is categorized into the parallelizing group. The compilation units in this group is performed parallelizing compilation. At the parallelizing compilation for the compilation unit, the compiler employs the analysis information from other compilation units categorized in the sequential group. If a function is worth employing interprocedural restructuring, such as inlining, for, its body should be defined in the compilation unit of this group. In other words, if a function that should be inlined and its caller function are defined in separate compilation units, they should be integrated at the compilation unit level or IR level.
  As mentioned in Section III-A, the OSCAR compiler can employ hierarchical parallel processing for the source program. When there are nested parallel processing parts across different compilation units, the program part having the highest parallelism is put in the parallelizing group. $Para$ calculated by Equation (1) can be used as a measure of this decision for the case of the OSCAR compiler. Considering the nested parallelism is a future work.
- **Sequential Group**
  Other compilation units than those in the parallelizing group is categorized in the sequential group. Program analysis, instead of parallelization, is performed on the compilation unit in this group. The compiler generates an analysis information file from it as well as its IR. A compilation unit in this group includes the body of functions called by the compilation units in the parallelizing group.

The next point of the proposed compilation flow is how the compilation units can be categorized into the above two groups. We prepare two approaches as the following:

- **Manual Grouping**
  The manual grouping is the simplest approach to employ the proposed method. In this approach, the application developers have a responsibility of appropriately categorizing the compilation units into either group based on the knowledge of the program design in terms of the parallelism in it.
  If the developers have enough knowledge both of the program design and parallel processing, this approach

can introduce the efficient parallel processing with small recompilation cost.

- **Automatic Grouping**
  Contrary to the manual grouping, the automatic grouping categorizes the compilation units into two groups automatically by the compiler. In this approach, first, all of the compilation units are integrated into a single file. Then, the compiler analyses the integrated program and detects the parallelism for each part of the program. According to the detected parallelism, the compiler categorizes the original compilation units into either group. For the case of the OSCAR compiler, $Para$ for each MTG calculated by Equation (1) is used in this approach; the compilation units having low para (usually less than 2) are put in the sequential group, and other units are put in the parallelizing group. Note that the threshold of $Para$ distinguishing these groups can be decided by the developers depending on the characteristics of the application and the computational resource in the target hardware.
  This approach does not require the developers to have the exact program design in terms of the parallel processing because the compiler fully has a responsibility of the grouping. On the other hand, this requires parallelism detection phase. However, all of the compilation units must be compiled at the first time of the program build time anyway. In addition, this parallelism detection and grouping are not required again at the recompilation time if the drastic modification, which results in the exploitation of more parallelism from a compilation unit in the sequential group, is not employed.
  There is another consideration in this approach when a compilation unit has two types of functions: the one is functions having high $Para$ and the another is those having little $Para$. When this compilation unit is involved in the parallelizing group, at least the file reading cost for the little $Para$ functions must be paid even if the compiler can stop the analysis on them by providing the analysis information. On the other hand, when the unit is involved in the sequential group, the parallelism of high $Para$ functions in this unit cannot be utilized at all. In this paper, because of the expected parallel processing performance of the compiled programs, we took the first approach such that the compilation units having high $Para$ functions are involved into the parallelizing group and the compiler stops the analysis on the little $Para$ functions by providing the analysis information of them.

B. Automatic Parallelizing Compilation by Proposed Flow

The proposed separate compilation flow deals with the previously explained parallelizing group and sequential group in different ways. Firstly, the compilation units of the sequential group are compiled and their analysis information files are exploited. Then, the compilation units of the parallelized group are compiled with the analysis information and parallelized with interprocedural analysis passes and restructuring passes.

Figure 3 and Figure 4 show the compilation flow for the sequential group and the parallelizing group, respectively. The compilation flow for each group consists of the front-end, the middle-end, and the back-end, as described in Section III-A. The front-end generates an IR from the compilation units for both cases.

For the compilation units of the sequential group, the middle-end generates an analysis information file from the compilation unit along with its IR file. Then, the back-end and the platform compiler generate the object file to be linked together with the parallelizing group from the IR file. Note that a compiler does not parallelize the compilation units of this group.

The current version of an analysis information file contains the following information for each function in a compilation unit as discussed in Section III-B:

- Defined variable list
- Used variable list
- Execution cost
- "sequential" flag described

The access range information for arrays will be added at the next version.

On the other hand, for the compilation units of the parallelizing group, the middle-end integrates analysis information files from the sequential group, and then performs parallelization on a compilation unit by utilizing integrated information. Finally, the back-end and the platform compiler generate the parallelized executable object file.

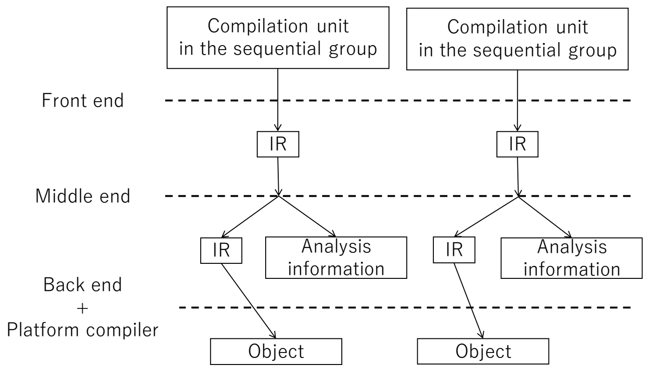
Compilation unit in the sequential group
Compilation unit in the sequential group
Front end
IR
IR
Middle end
IR
Analysis information
IR
Analysis information
Back end + Platform compiler
Object
Object

Fig. 3. Overview of the compile flow for the sequential group. The middle-end generates the analysis information file and IR for each compilation unit.

Note that an analysis information file includes the information discussed in Section III-B for each function defined in a compilation unit.

C. Recompilation with Proposed Compilation Flow

Recompilation is occurred when one or some of compilation units in a source program are modified. If a compilation unit in the parallelizing group is modified, as expected, the compiler performs parallelizing compilation on it again. On the other hand, if a compilation unit in the sequential group is modified, the compiler performs compilation on it again, and generates the updated analysis information file and IR file; then, the

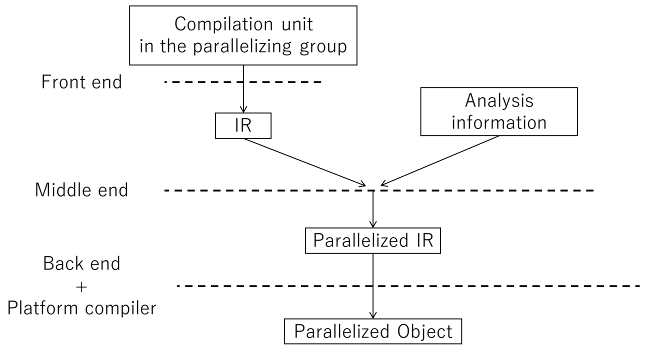

Fig. 4. Overview of the compile flow for the parallelizing group. The middle-end analyzes the parallelizing group by using IR from the parallelizing group and analysis information from the sequential group.

compilation units of the parallelizing group calling functions in the modified compilation unit are also compiled again. Furthermore, if a function called by a compilation unit in the parallelizing group is added, it is inserted in a compilation units of the sequential group and compiled; then, a compilation unit of the parallelizing group calling the added unit is also compiled.

V. Experimental Evaluation

The proposed separate compilation flow is implemented in the OSCAR automatic parallelizing compiler. We evaluate it by using four real control programs from industry.

A. Evaluation Environment and Programs

We use a dual CPU socket server machine with Intel Xeon E5-2637 v4 driven at 3.5 GHz(4 cores per one socket, no hyper-threading) and 755.8 GB memory for the evaluation of compilation. Ubuntu 14.04 LTS 64bit is installed on it.

Four huge sized real control programs from industry are used for the evaluation; they are labeled from "Program1" to "Program4". They constitute a large real-time system having several deadlines. The purpose of parallelizing them is reducing their execution time as much as possible to enable the system developers to implement more functions within the predefined deadlines. One of their important characteristics is the little loop iteration level parallelism in them. Therefore, ordinary parallelizing compilers cannot exploit parallelism from them. On the other hand, there is sufficient coarse grain task parallelism that the OSCAR compiler can exploit [14].

Table I shows the summary of the evaluation programs in terms of their scale. The "operation" in this table stands for the IR-level primitive operations like the LLVM-instructions [18]. The compiler traverses them repeatedly at the analysis and restructuring time, thus the number of them affects the total compilation time. According to the table, those programs take 64.24 seconds, 71.16 seconds, 6,973.45 seconds, and 3,755.35 seconds, respectively for the compilation by the OSCAR compiler when all the compilation units are integrated in a single file. Each program is reorganized into the parallelizing group and the sequential group through the proposed compilation flow. This table also shows the ratio of operations in the parallelizing group for each program. "(manual)" and "(auto)" stand for the manual and the automatic grouping, respectively. Before the parallelizing compilation, the task execution cost for the programs is measured on the target multicore controller, and the compiler employs it to ensure accurate task scheduling.

B. Results

Under the assumption of the recompilation scenario, we first evaluate the compilation time of the parallelizing group of the proposed compilation flow (with code separation) as well as the time taken by a single file integrating all the compilation units of a program (without code separation). For the proposed compilation flow, the manual grouping (manual) and the automatic grouping (auto) are also evaluated. Note that the cases of "without code separation" correspond to the IR-based LTO described in Section II-A.

Figure 5 shows the result of the evaluation. For all the programs, the compilation time can be reduced by the proposed compilation flow. Compared with the compilation without the proposed flow, which employs parallelizing compilation to the single file combining all compilation units, the proposed technique with manual grouping reduces the compilation time to 44%, 53%, 4%, and 22%, respectively. Similarly, the proposed technique with automatic grouping reduces it to 51%, 65%, 10%, and 53%, respectively. The reduction in the compilation time for each program corresponds to the ratio of operations in the parallelizing group, as shown in Table I. The compilation time for the automatic grouping includes the file reading time for the non-parallelized program part compared with the manual grouping. Hence, the compilation time for the automatic grouping is longer than the manual grouping. This can be reduced, for example, by removing the non-parallelized program part from the parallelizing group.

Figure 6 shows the estimated performance of the parallelized programs on an infinite number of processor cores. The compiler calculates them by dividing the sequential program execution cost by the critical path length of its MTG with its task execution cost obtained by profiling as same as $Para$ in Section III-A. The four bars for each program are the results of the compilation on the single file combining all the compilation units (without code separation), the separate compilation with the proposed flow and manual grouping (with code separation (manual)), the separate compilation with the proposed flow and automatic grouping (with code separation (auto)), and the separate compilation without the analysis information file (separation compilation without analysis information file), respectively. They are normalized so that the estimated performance without the code separation becomes 100%. The reason for showing the estimated performance instead of the performance on the target multicore is to clarify the compiler's ability to exploit parallelism.

This figure shows that the proposed flow obtains better performance for all the programs in the cases of both the manual and the automatic grouping than the cases of the separate compilation without the analysis information file.

TABLE I
SUMMARY OF THE EVALUATED PROGRAMS

	Number of operations	Number of variables	Number of modules	Number of files	Compilation time[s]	% of operations in parallelizing group	
						(manual)	(auto)
Program1	62,821	4,051	988	212	64.24	30.91	30.91
Program2	35,411	2,953	912	235	71.16	41.54	41.54
Program3	326,612	15,780	6,751	1,226	6,973.45	12.87	12.06
Program4	534,933	21,284	8299	1,520	3,755.35	19.72	27.28

Specifically, it has 26% better performance for Program4 (74% $\rightarrow$ 100%) with both grouping approaches. The compiler can exploit more parallelism among tasks including function calls by employing interprocedural analysis information. Interestingly, the automatic grouping can obtain better performance than the manual grouping for Program3 (92% $\rightarrow$ 99%). The compiler can find more parallelizable part from the program than the program developer for this case.

Compared with the performance of the cases without separate compilation, the proposed flow obtains comparable performance for Program1, 3, and 4. However, for Program2, the proposed flow shows 86% of performance. This performance degradation is due to the insufficient representation of pointer variables and structure members in the current version of the analysis information file. The current version does not support the information of pointer initialization statements. It also does not deal with structure members inside deeply nested structures. Thus, the compiler cannot handle them in other compilation units, even with the analysis information file. These points can be fixed at the next version.

It is usual that only a small part of the program is modified in program debugging and performance tuning phases. When all the IR are combined or read at the recompilation time, as in the case with the IR-based LTO, the compilation time is gradually increased because reading the IR requires costly parsing and program analysis passes at the link phase, in addition to recompilation time for the modified compilation units. Instead, the proposed compilation flow takes analysis information files for the compilation units of the sequential groups; consequently, it requires small reading cost and no analysis cost in addition to the compilation time for the parallelizing group. To clarify the additional recompilation cost for the compilation units of the sequential group, we measure their compilation time, as shown in Table II. According to the table, the compilation time is 0.6 seconds to 1.7 seconds on average, and this is paid at the recompilation of the modified compilation unit. Thus, the proposed compilation flow can reduce the recompilation time.

VI. CONCLUSION

While there is no doubt about the importance of preoccupation in the performance of a program, its maintainability and compilation time are also important factors of software development, especially for large scale practical programs. Separate compilation has been widely used to ensure the later factors. On the other hand, the scope of a compiler optimization is usually restricted by the compilation unit; thus, the opportunity for the performance improvement is reduced. Link Time Optimization (LTO) has been used to overcome this problem. However, previously proposed LTO techniques are difficult to directly employ for automatic parallelization, or they introduce longer compilation time.

In this paper, we proposed a separation compilation technique for automatic parallelization. In this flow, the compila-

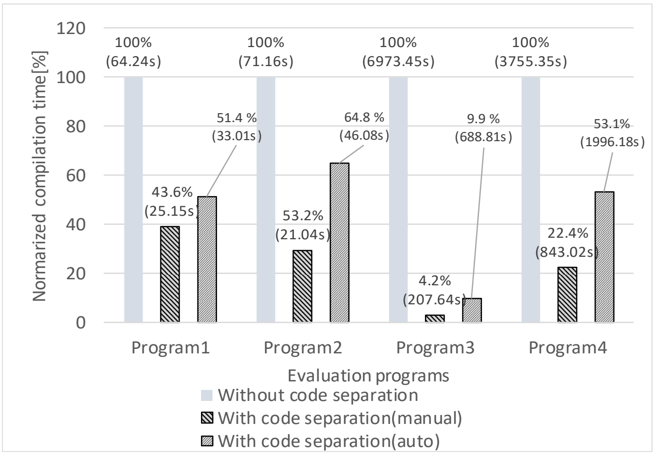

Fig. 5. Compilation time measurement result. Each bar shows the normalized compilation time against the case for combining all compilation units into a single file. “manual” and “automatic” stand for the manual grouping and the automatic grouping, respectively.

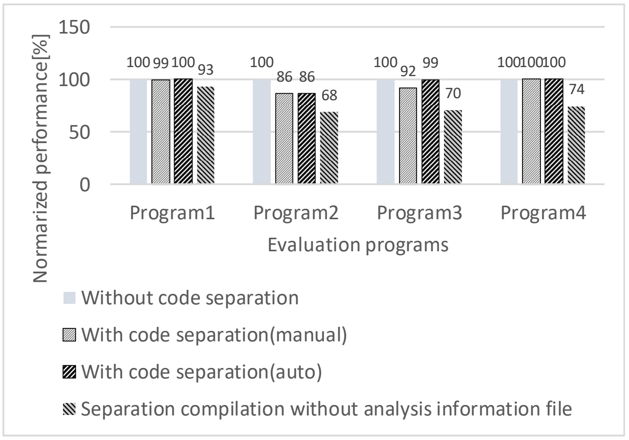

Fig. 6. Estimated performance for the programs. Each bar shows the normalized performance against the case for combining all compilation units into a single file.

TABLE II
COMPILATION TIME FOR THE SEQUENTIAL GROUP

	Number of compilation units	Average compilation time for sequential units[s]	Standard Deviation of compilation time for sequential units [s]
Program1	95	0.6	1.7
Program2	28	0.8	0.9
Program3	471	1.7	12.2
Program4	883	0.8	2.1

tion units of a source program are categorized into the parallelizing group and the sequential group. This grouping can be employed either manually by the developers or automatically by the compiler. Then, the analysis information is exploited from each compilation unit of the sequential group. The compiler performs parallelization on the parallelizing group by integrating the analysis information from the sequential group. Thus, the compilation flow achieves both full parallelization of a program and shorter compilation time.

The proposed compilation flow is implemented on the OSCAR automatic parallelizing compiler. The experimental evaluation result using large real control programs from industry shows the proposed method can achieve 29% better performance than the separate compilation without the proposed method, and the compilation time can be reduced by 90% with only 1% of performance degradation, compared with the compilation of the fully unified source code into a single compilation unit when the automatic grouping is employed.

REFERENCES

[1] Cilio, A.G.M., Corporaal, H.: Link-time effective whole-program optimizations. Future Generation Computer Systems(FCGS), 2000, 16(5), pp. 503-511, March 2000.

[2] Johnson, T., Amini, M., Li, X.D.: ThinLTO: Scalable and Incremental LTO. 2017 IEEE/ACM International Symposium on Code Generation and Optimization(CGO), February 2017.

[3] Kasahara, H., Honda, H., Aida, K., Okamoto, M., Yoshida, A., Ogata., W.: OSCAR Fortran Multigrain Compiler. In Parallel Language and Compiler Research in Japan, Springer, pp271-301, 1995.

[4] Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S., Tseng, C., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. ACM SIGPLAN Notices, 1994, 29(12), pp. 31-37, Dec 1994.

[5] GNU Compiler Collection (GCC) Internals: LTO Overview, https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html#LTO-Overview.

[6] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Mar. 2004.

[7] Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: OSCAR API for Real-Time Low-Power Multicores and Its Performance on Multicores and SMP Servers. International Workshop on Languages and Compilers for Parallel Computing. (LCPC) 2009, Springer, pp. 188-202, 2010.

[8] OpenMP. https://www.openmp.org/.

[9] OpenACC. https://www.openacc.org/.

[10] NVIDIA CUDA Programming Guide. https://docs.nvidia.com/cuda/.

[11] Grosser, T., Groesslinger, A., Lengauer, C.: Polly - Performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters. 22(4), 2012.

[12] Ayers, A., Jong, S.D., Peyton, J., Schooler, R.: Scalable cross-module optimization. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation(PLDI'98), pp. 301312, 1998.

[13] Moon, S., Li, X.D., Hundt, R., Chakrabarti, D.R., Lozano, L.A., Srinivasan, U., Liu, S.M.: Syzygy - a framework for scalable cross-module ipo. In Proceedings of the International Symposium on Code Generation and Optimization: Feedbackdirected and Runtime Optimization(CGO04), 2004.

[14] Umeda, D., Suzuki, T., Mikami, H., Kimura, K., Kasahara, H.: Multigrain Parallelization for Model-Based Design Applications Using the OSCAR Compiler. International Workshop on Languages and Compilers for Parallel Computing(LCPC) 2015, Springer, pp. 125-139, 2015.

[15] Obata M., Shirako J., Kaminaga H., Ishizaka K., Kasahara H. (2005): Hierarchical Parallelism Control for Multigrain Parallel Processing. In: Pugh B., Tseng CW. (eds) Languages and Compilers for Parallel Computing. LCPC 2002. Lecture Notes in Computer Science, vol 2481. Springer, Berlin, Heidelberg.

[16] Han, J., Fujino, R., Tamura, R., Shimaoka, M., Mikami, M., Takamura, M., Kamiya, S., Suzuki, K., Miyajima, T., Kimura, K. and Kasahara, H. Reducing Parallelizing Compilation Time by Removing Redundant Analysis. Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems(SEPS2016), pp.1-9, 2016.

[17] Y. Yu, H. Dayani-Fard, J. Mylopoulos and P. Andritsos, "Reducing build time through precompilations for evolving large software," 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 2005, pp. 59-68.

[18] LLVM Language Reference Manual LLVM 9 documentation. https://llvm.org/docs/LangRef.html.