Reducing Parallelizing Compilation Time by Removing Redundant Analysis

Jixin Han Rina Fujino Ryota Tamura Mamoru Shimaoka Hiroki Mikami
Waseda University, Japan
{kalfazed,rfujino,r_tamura,shimaoka,hiroki}@kasahara.cs.waseda.ac.jp

Moriyuki Takamura Sachio Kamiya Kazuhiko Suzuki Takahiro Miyajima
OSCAR TECHNOLOGY CORPORATION, Japan
{takamura,kamiya,suzuki,miyajima}@oscartech.jp

Keiji Kimura Hironori Kasahara
Waseda University, Japan
{keiji,kasahara}@waseda.jp

Abstract

Parallelizing compilers employing powerful compiler optimizations are essential tools to fully exploit performance from today's computer systems. These optimizations are supported by both highly sophisticated program analysis techniques and aggressive program restructuring techniques. However, the compilation time for such powerful compilers becomes larger and larger for real commercial application due to these strong program analysis techniques. In this paper, we propose a compilation time reduction technique for parallelizing compilers. The basic idea of the proposed technique is based on an observation that parallelizing compilers apply multiple program analysis passes and restructuring passes to a source program but all program analysis passes do not have to be applied to the whole source program. Thus, there is an opportunity for compilation time reduction by removing redundant program analysis. We describe the removing redundant program analysis techniques considering the inter-procedural propagation of analysis update information in this paper. We implement the proposed technique into OSCAR automatically multigrain parallelizing compiler. We then evaluate the proposed technique by using three proprietary large scale programs. The proposed technique can remove 37.7% of program analysis time on average for basic analysis includes def-use analysis and dependence calculation, and 51.7% for pointer analysis, respectively.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Language-compilers

SEPS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4641-2/16/11...$15.00
http://dx.doi.org/10.1145/3002125.3002129

Keywords parallelizing compiler; program optimizations; program analysis.

1. Introduction

The performance of a computer system strongly depends on the capability of its compiler optimizations. A number of sophisticated program analysis techniques and restructuring techniques have been developed to realize powerful optimizations especially for parallelizing compilers. However, these powerful optimization techniques require much compilation time to attain higher performance on a target machine.

Program developers may avoid using time consuming compiler optimizations to improve the program productivity even they can introduce higher performance to their programs. In order to reduce compilation time especially for large scale programs, several researches have tackled the large compilation time problem. Mehta et al. tries to reduce the number of statements in a source program by combining several statements, which have data dependence each other, into one statement to reduce the information for program analysis [1]. Nick et al. propose an efficient computation technique for a directed acyclic graph (DAG) of strongly connected components (SCC) [2]. A DAG of SCC contains fundamental information for program analysis techniques, thus this contributes to the compilation time. Yu et al. propose a precompilation technique that remove the false dependencies among the files to reduce the total build time of a target program [14].

In this paper, we propose a compilation time reduction technique by removing redundant program analysis especially for parallelizing compilers that usually apply aggressive program restructuring techniques to a target program. We also implement the proposed technique into the OSCAR automatically multigrain parallelizing compiler [4, 7].

OSCAR compiler can exploit parallelism from the whole program by applying multigrain parallel processing. In order to exploit parallelism and improve efficiency of paral-

lel execution at runtime, the compiler applies program restructurings multiple times according to the compiler options given by the user. After each program restructuring, the compiler must apply program analysis for the whole program again since the program restructuring may change a control flow graph (CFG) and a call-graph, add new variables, and so on. This means the compilation time increases as the program size increases when the compiler applies program analysis and restructuring multiple times. For instance, a program with 51,799 lines spends 2 hours for compilation. However, those restructuring passes do not always change the whole part of the program. Some functions (modules or procedures) may keep the original structure after a program restructuring.

Our proposed technique consists of two phases. At the restructuring time, the compiler records functions whose information including program structure is changed by the restructuring. Then, at the following analysis time, the compiler applies program analysis only to the recorded functions at the restructuring time to update the program analysis information. The previous analysis information of remained functions do not need to be updated since there is no change at the restructuring time.

One of the important points of this technique is we must take care of the propagation of the modification effects by a restructuring for inter-procedural analysis such as pointer analysis [9, 10]. For an inter-procedural analysis, the analysis information is propagated among functions. Therefore, the compiler must detect the functions that the analysis update is propagated, then the compiler also applies the analysis to these functions in addition to restructured functions. This paper describes how to deal with such inter-procedural update of analysis information in the proposed technique.

The main contributions of this paper are as follows:

- We investigate the organization of the parallelizing compiler and the time consumption for each part in the compiler.
- We then propose a compilation time reduction technique by removing redundant program analysis considering the propagation of analysis information among functions based on the above investigation result.
- We show the evaluation result of the proposed technique by using the large scale real applications.

The rest of this paper is organized as following: Section 2 provides the overview of the OSCAR compiler and its organization from the point of the restructuring and analysis in it. Section 3 investigates the analysis passes in the compiler. Section 4 describes the proposed compilation time reduction technique based on the investigation of Section 3. Section 5 shows the performance evaluation results. Section 6 introduces some related works. Section 7 finally concludes this paper.

2. OSCAR Automatically Parallelizing Compiler

We provide an overview of OSCAR compiler in this section. Then, we briefly describe several restructuring passes in the compiler. We also explain the relationship between restructuring passes and analysis passes in the compiler to show the motivation of this work.

2.1 Overview of OSCAR Compiler [4, 7]

OSCAR compiler parallelizes input C or Fortran77 programs by the multi-grain parallel processing, which consists of coarse grain task parallel processing among loops and function calls, near fine grain parallel processing among statements inside a basic block in addition to ordinary loop iteration level parallel processing.

In coarse grain task parallel processing, a source program is decomposed into three kinds of blocks, namely BB, SB and RB. Here, BB is a basic block, SB is a subroutine call, and RB is a repetition block, or an outer-most loop, respectively. These blocks are defined as macro-tasks (MTs). The compiler analyzes control flow and data dependence among MTs. The analysis result is represented as a macro-flow-graph (MFG). Then, the compiler exploits the parallelism among MTs from an MFG by applying the earliest executable condition analysis [7] and represents it as a macro-task-graph (MTG). Finally, macro-tasks are assigned to cores statically when there is no conditional branches among macro-tasks. Otherwise, the compiler embeds dynamic scheduling code inside the parallelized program. If an RB or an SB has coarse grain task parallelism inside it, the compiler hierarchically decomposes the body of it into macro-tasks and generate an MTG for them. Those SB and RB applied coarse grain task parallel processing are called as "parallel processing layers" [5].

2.2 Restructuring and Analysis in OSCAR Compiler

OSCAR compiler tries to exploit coarse grain task parallelism from the whole program. Therefore, it globally applies the program analysis to the source program. In addition, the compiler applies several restructuring in order to exploit more parallelism among macro-tasks and also to reduce the runtime overhead. Here, we explain two of restructuring in the compiler as examples: inline expansion and macro task fusion. They may dramatically change the program structure and information of modules such as CFG, number of variables, the points-to information of pointer variables, and so on.

2.2.1 Inline Expansion

Inline expansion is one of conventional and popular compiler optimizations. It reduces the function call overhead by embedding the body of the callee function at the call-site.

It can be also used for exploiting more coarse grain task parallelism by merging parallelism inside the function body

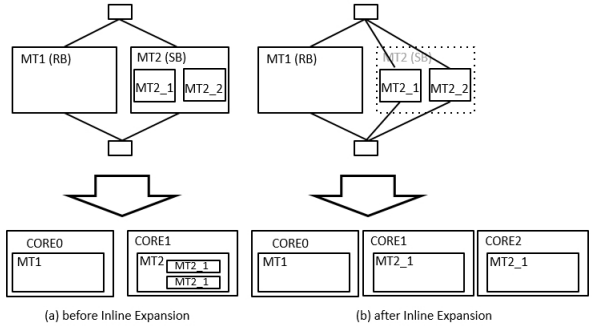


Figure 1. Exploiting parallelism by Inline Expansion. There are MT1 and MT2 in the same parallel processing layer. MT2 also has MT2_1 and MT2_2 and there are no dada dependence among MT1, MT2_1and MT2_2. Before inline expansion (a), only parallelism between MT1 and MT2 can be exploited. By applying inline expansion to MT2, all parallelism among MT1, MT2_1 and MT2_2 can be exploited.

into the parallelism at the call-site [11]. For instance, assume that there are MT1 and MT2 inside a program and MT2 is a SB (function call), and they can be processed in parallel (Figure 1). In addition, MT2_1 and MT2_2 also exist inside MT2, and there are no data dependence among MT1, MT2_1 and MT2_2. When inline expansion is not applied to MT2, only parallelism among MT1 and MT2 is available since each core is assigned onto one core. In contrast, the parallelism among MT1, MT2_1 and MT2_2 is available by applying inline expansion since they can be placed on the same parallel processing layer and assigned to distinguished cores simultaneously.

When inline expansion is processed, OSCAR compiler traverses the call-graph of the source program and checks each call-site in the graph whether its body can be expanded, or not. All blocks and variables in an expanded function are merged into the call-site, then the compiler regenerates the call-graph.

2.2.2 Macro Task Fusion [12]

Macro task fusion is a restructuring technique for coarse grain task parallel processing. It merges multiple macro-tasks into one macro-task to minimize scheduling overhead.

As described in Section 2.1, OSCAR compiler applies dynamic scheduling to an MTG when it has conditional branches among macro-tasks. However, the granularity of macro-tasks tend to be relatively smaller than the dynamic scheduling overhead especially for the cases of control applications [12]. The efficiency of parallel processing becomes worse for such a situation. By applying macro task fusion, conditional branches inside an MTG can be hidden inside macro-tasks. Thus, static scheduling can be applied to the MTG and the scheduling overhead becomes minimal.

The compiler processes macro task fusion as the following steps: First, the compiler traverses macro-tasks inside an MTG to check conditional branches. When a conditional branch is found, the compiler checks post-dominated macro-tasks by this branch. These macro-tasks are recorded as a group to be merged into one macro-task. Then, macro-tasks inside this group is merged into a new macro-task. This restructuring modifies CFG of a source program. Therefore, the compiler generates an MTG again after macro task fusion.

2.2.3 Ordering of Analysis-passes and Restructuring-passes

OSCAR compiler applies multiple kinds of restructuring to a source program to fully exploit coarse grain parallelism. Each restructuring requires program analysis information. If the compiler applies restructuring and modifies program structure, it must also applies program analysis again.

Figure 2 shows an example compilation flow including multiple restructuring passes and analysis passes. In this flow, the compiler applies inline expansion firstly to exploit coarse grain task parallelism from a source program as described in Section 2.2.1. After inline expansion, the compiler applies a sequence of program analysis.

Then, the compiler generates MTGs for RBs and SBs by utilizing program analysis information, and determines parallel processing layers by evaluating parallelism of generated MTGs [5].

Next, the compiler applies macro task fusion to each MTG. A sequence of program analysis follows it since it may modify the program structure. Finally, the compiler generates MTGs by using updated program analysis information, then generates the parallelised program.

Note that the compiler applies a sequence of program analysis to the whole program even at the second time after macro task fusion to update the analysis information. This is because the structure of the program may be changed in different extent after macro task fusion. However, some functions do not include MTGs when they do not have enough coarse grain task parallelism. Furthermore, the compiler does not change an MTG when it does not have any conditional branches. It means we have an opportunity to reduce program analysis time by skipping program analysis for functions that is not modified by the previous program restructuring.

3. Investigation into Analysis-passes

3.1 Time consumption of Analysis-passes

Table 1 show the compilation time of OSCAR compiler for three real proprietary large scale programs: Control Program A, Control Program B and Image Processing Program. The detail information of these programs will be shown in Section 5.2.

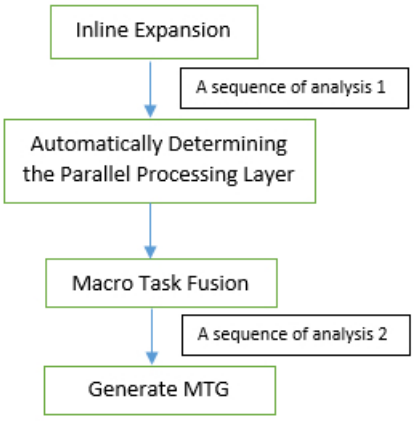


Figure 2. An example of an order of analysis passes and restructuring passes in OSCAR compiler.

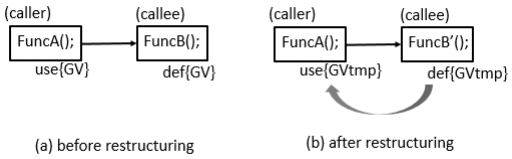


Figure 3. Example of interprocedural effect of program restructuring. Before restructuring (a), FuncB() defining (modifying) a global variable 'GV' is called by FuncA() using it. After restructuring (b), FuncB'() defines a variable 'GVtmp' instead of 'GV'. The analysis result of FuncA must be 'useGVtmp' to reflect the restructuring result.

As shown in the table, there are main time consuming passes in the compiler such as pointer analysis, def-use analysis, dependence computation and restructuring processes, though their proportion depends on the applications. As discussed in Section 2.2.3, there is an opportunity to reduce analysis time for pointer analysis, def-use analysis and dependence computation. Especially, pointer analysis occupies 50% of the compilation time for the case of Control Program A. We focus on these three analysis passes in this section.

For these analysis, we must take care of the propagation of analysis information update among functions in a source program to remove redundant analysis appropriately. For instance, assume that a function FuncA() calls another function FuncB() (Figure 3). If FuncB() defines a global variable GV, the information of this variable definition is propagated to the call-site FuncA(). If a restructuring pass modifies FuncB(), that definition information may be changed and this information update must be propagated to the FuncA(). Thus, the compiler must apply program analysis passes to FuncA() even when it is not modified by the restructuring pass.

We briefly review the overview of those three analysis passes in the rest of this section.

3.2 Def-Use Analysis

Def-use analysis is one of the fundamental analysis passes in OSCAR compiler. In def-use analysis, the compiler traverses all statements in a source program and records defined- and used- variables and arrays for each statement. If a statement is a function call, def- and use- information inside that function is merged to that function call statement. The results of def-use analysis pass are later used in various other program analysis passes in the compiler, such as data-flow analysis, array access pattern analysis [6], parallel-loop analysis [3], and so on.

3.3 Dependence Calculation

In dependence calculation, the compiler calculates data dependencies among macro-tasks for each parallel processing layer in a source program based on the results of def-use analysis and other related data access analysis passes. The compiler checks data dependence between one macro-task and other all macro-tasks in the same parallel processing layer in a function. Thus, if there are n macro-tasks in a function, it takes the cost of $O(n^2)$. It means the analysis cost of dependence calculation becomes larger along with the size of a source program. Note that the def- and use- information within a function called from a macro-task, of course, affect dependence calculation here. We must also consider the inter-procedural information propagation.

3.4 Pointer Analysis

The pointer analysis pass calculates points-to information that represents the relationship between pointer variables and pointed object from those pointer variables. Flow-sensitive and context-sensitive pointer analysis [9, 10] is employed in OSCAR compiler.

In pointer analysis, the compiler firstly makes initial points-to information for each basic block by checking assignment statements for pointer variables. Then, the points-to information is propagated along with the control flow graph (CFG) in a source program considering conditional branches and loop structures by well-known data-flow analysis manner. The analysis result is recorded as in- and out-points-to information for each basic block, loop block and function module. Here, function calls are also taken into account to propagate the points-to information over functions. In order to propagate information, invocation graph, which is constructed from a call-graph of a source program, is used for context-sensitive pointer analysis [9]. In an invocation graph, each call-site is represented as a different node to different call-sites that invokes the same function. The compiler integrates the points-to information of caller functions

into the call-site along with the invocation-graph. It means the update of the points-to analysis information in one function clearly affects the other functions. In addition, the result of pointer analysis is widely used in other analysis passes including def-use analysis.

4. Proposed Compilation Time Reduction Technique

4.1 Overview

As discussed in Section 2.2.3, OSCAR compiler applies multiple program analysis passes and restructuring passes to a source program. Even after the second analysis time, the compiler applies it to the whole program no matter if all functions are changed by the restructuring passes, or not. However, if a function is not modified by a restructuring pass, the compiler does not have to apply a program analysis to the function again after that restructuring pass. Thus, we can reduce the compile time by removing such redundant program analysis.

We introduce a bitvector data structure in the compiler. At a restructuring pass, the compiler records modified functions in the bitvector by their ID numbers starting from 1. For instance, if functions of 2, 3, 5, 7 and 10 are modified in macro task fusion explained in Section 2.2.2, the compiler records them as the following:

$$BitVector = \{0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, ...\}$$

The program analysis passes following the restructuring pass decide whether re-analysis must be applied or not by checking the bitvector.

We describe how the compiler removes redundant analysis by utilizing the bitvector in the rest of this section.

4.2 Removing Analysis Considering Call-graph

The compiler traverses functions in a source program at a program analysis time. The basic idea of the proposed compilation time reduction technique in this paper is skipping the analysis for unmodified function at the previous restructuring pass by checking the bitvector as described in Section 4.1.

For this bitvector checking, we must take care of the inter-procedural update of analysis information. The compiler realizes this by checking a call-graph in addition to the bitvector as following:

- If the bitvector indicates the processing function is modified, the compiler applies the analysis for it.
- If the bitvector indicates any child function (callee-function) of the processing function is modified, the compiler also applies the analysis for it.
- Otherwise, the compiler can skip the analysis.

Thus, the compiler can skip the redundant analysis considering the inter-procedural analysis update.

4.3 Removing Analysis Considering Global Effects by Global Scope Objects

Objects in a program having global scope, such as global variables and heap objects [8], may affect multiple functions even when they have no direct relationship in a call-graph. This must be taken into account for program analysis passes employed by data-flow analysis manner like the pointer analysis pass. Figure 4 shows an example of such a case in pointer analysis.

In this figure, a global pointer variable points to some memory location. This points-to relationship is made in Block-A of a Function1 and it is used by Block-B of Function2. In other words, the "out" set of Block-A and Function1 made by the pointer analysis includes this points-to information, and the "in" set of Block-B and Functin2 also includes it, respectively. When this out-set is changed by the modification of Function1 in the previous restructuring pass, the in-set is also changed. Therefore, the compiler must apply analysis passes again on Function2.

Here, assume that Function1 and Function2 are called by another function, namely Function3, while Function1 and Function2 are neither callee- nor caller- function each other. Thus, there is no parent-child relationship between Function1 and Function2 in a call-graph. The rules discusses in Section 4.2 is not enough since they relies on the parent-child relatinship in a call-graph. The following rule is added to deal with global effects caused by global scope objects:

- If the in-set of the processing function has global scope objects analyzed by the previous analysis pass, the compiler checks whether the function generating the out-set that includes those global scope objects are modified by checking the bitvector. If modified, the compiler applies the analysis again for the processing function.

This can be extended to the analysis process of traversing blocks (basic blocks, loops, function call statements), in a function module to reduce analysis time further. When the compiler traverses the blocks, the compiler also checks the global scope objects in the in-set of them. If the in-set includes the compiler also checks the modification of functions that generates corresponding out-set. If there is no modification in them, the compiler skips data-flow calculation for the processing block.

4.4 Additional Modification for Pointer Analysis Pass

While the modification for the pointer analysis pass is basically same as that for other analysis passes, we must take care of construction of an invocation graph for pointer analysis.

The compiler constructs an invocation graph from a call-graph before pointer analysis. Different call-sites in a function are represented as different nodes in an invocation graph so that the compiler can distinguish different contexts, or

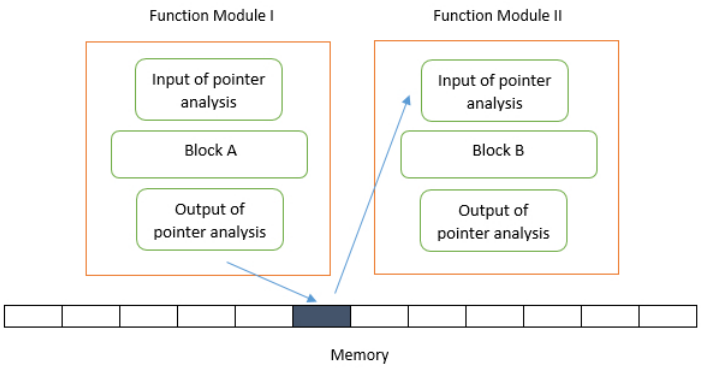


Figure 4. Example of global effects among functions by global scope objects. A global pointer variable modified in Block-A of Function1 is used in Block-B of Function2. Assume these two functions have no direct relationship in a call-graph.

pointer-analysis information, on those call-sites. The pointer analysis pass uses it to propagate points-to information.

If a restructuring pass like inline expansion modifies the call-graph, the invocation graph must be totally constructed again. However, other almost restructuring passes do not modify the call-graph. Therefore, the compiler can only take care of modified functions recorded in the bitvector to reconstruct the invocation graph.

5. Experimental Evaluation

We implemented the proposed compilation time reduction technique in OSCAR compiler. We evaluated it using three proprietary large scale programs.

5.1 Evaluation Environment

We used Intel Xeon E5-2667 v4 for our evaluation. The Xeon E5-2667 v4 processor has 16 cores driven at $2.30GHz$. The evaluated machine has 512GB memory. We used Ubuntu 14.04 LTS 64 bit as the operating system.

5.2 Evaluation Programs

We used three large-scale proprietary applications for the evaluation. Two of them are control programs for mechanics, namely "Control Program A" and "Control Program B", and the rest one is an image processing program, namely "Image Processing". We chose these three because of its long compilation time. At the same time, in order to compare the compilation time and the characteristics of program structure for them, we gave the same compiler options to the compiler.

Table 1 shows the analysis time and restructuring time consumption in seconds for the evaluation programs before implementing the proposed technique. Similarly, Table 2 shows the basic information of program structure for the programs.

From these tables, Control Program A spends $2519[sec]$ for its compilation. The most time consuming part for this program is the pointer analysis, which spends $1259[sec]$ and it is nearly 50% of the total compilation time. About the def-use analysis, it takes $201[sec]$ and it is about 8%. Regarding to the program structure, Control Program A has 83 functions, 3,833 variables, 536 pointer variables and 584 structures.

On the other hand, Control Program B has 675 modules and 1,046,244 variables, 164 pointers and 95,867 structures. The total compilation time for it is $104[sec]$. Def-use analysis requires $10[sec]$ and it is about 9%. Pointer analysis uses $12[sec]$ and it is nearly 10%.

Finally, for Image Processing, it has 173 modules, 12,851 variables, 689 pointers, and 8,873 structures, respectively. The total compilation time is $16.62[sec]$. In this program, the pointer analysis and def-use analysis will not use so much time The pointer analysis requires about $2.43[sec]$ (14%), and the def-use analysis spends $1.87[sec]$ (11%).

We can find the program analysis passes consume over 50% of the total the compilation time though the percentage of them are different depending on the applications. Especially, three main analysis passes discussed in Section 3 take at least 30% of the time.

5.3 Evaluation result

Figure 5 shows the evaluated compilation time reduction results by the proposed technique. In figure5(a), we categorized and evaluated analysis passes into two groups: the basic analysis pass and the pointer analysis pass. The basic analysis pass includes the def-use analysis pass and the dependence calculation pass. "Before" and "After" stand for the OSCAR compiler without and with the proposed technique, respectively. Each bar in the graph shows the normalized analysis time for each analysis group of each evaluated program based on OSCAR compiler without proposed technique. Figure5(b) shows the changes of the total compilation time.

For all programs and program analysis groups, the analysis time can be reduced by the proposed technique. Compared with the compiler without the proposed technique, the proposed technique attains 35% of time reduction for the basic analysis and 43% for the pointer analysis, respectively, for the case of Control Program A. Similarly, 41% and 59% time reduction are achieved for the basic analysis and the pointer analysis, respectively, for Control Program B. 37% and 53% time reduction are also obtained for Image Processing program.

The proposed technique in this paper aims at reducing compilation time with preserving same optimization strength. Therefor, the generated parallelised programs by the compiler with proposed technique are totally same as those by the compiler without the technique.

We discuss the evaluation result here focusing on the time reduction results of the pointer analysis for Control Program A.

	Total compilation time	Pointer analysis	Def-use analysis	Dependence calculation	Other analysis	Restructuring
Control Program A	2,519 (100%)	1,259 (50%)	201 (8%)	251 (10%)	377 (15%)	428.23 (17%)
Control Program B	104 (100%)	12 (12%)	10 (9%)	9 (9%)	21 (18%)	53 (51%)
Image Processing	16.62 (100%)	2.43 (14%)	1.87 (11%)	3.86 (23%)	3.15 (19%)	5.48 (33%)

Table 1. Analysis time and restructuring time for each program in seconds.

	Num. of lines	Num. of functions	Num. of variables	Num. of pointers	Num. of structures
Control Program A	334,297	83	3,833	536	584
Control Program B	1,720,919	675	1,046,244	164	95,867
Image Processing	173,596	173	12,851	689	8,873

Table 2. Information of evaluated programs.

The sequence of the analysis in this evaluation includes the pointer analysis, the def-use analysis, the dependence calculation, and an ordinary control-flow analysis. The restructuring process includes the inline expansion and the macro task fusion explained in Section 2. The macro task fusion consists of pre-process for finding conditional branches and the fusion process itself.

The compiler carries out all analysis to all functions in a source program at the first sequence of the analysis. Then, as discussed in Section 4, the compiler can re-use the analysis results at the rest of the analysis sequences if the functions related to the processing function are not modified. Here, the analysis time for other analysis sequences than the first analysis clearly depends on the number of functions modified by the restructuring passes.

For the case of the pointer analysis pass, the compiler applies it to a source program five times in this evaluation. The first analysis is carried out to 94 functions for Control Program A while the second analysis after the inline expansion pass is applied to zero function since the inline expansion is not applied to any function in this case. The fourth and fifth analysis passes are applied after the pre-process of the macro task fusion and the body of the macro task fusion, respectively. The number of modified functions by these restructuring passes is 53. Therefore, nearly half of the functions are not modified and the compiler can skip the analysis for them. Thus, the proposed technique can reduce 43% of the analysis time for the pointer analysis for Control Program A.

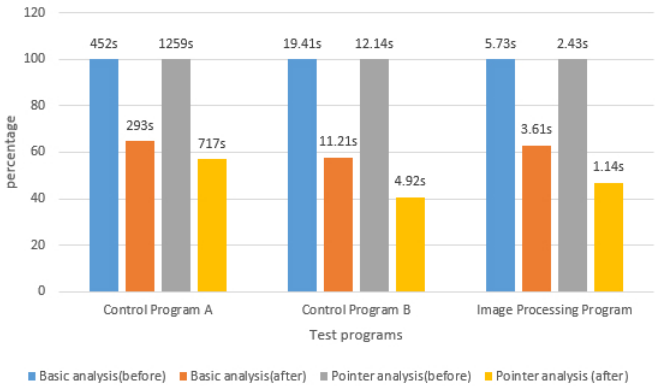
6. Related Works

Several literature have dealt with the issues related to the reduction of compilation time and the scalability of compilers for large scale programs. Some past works focused on tackling the compilation time by exploiting the data dependency or bundle properties from the program.

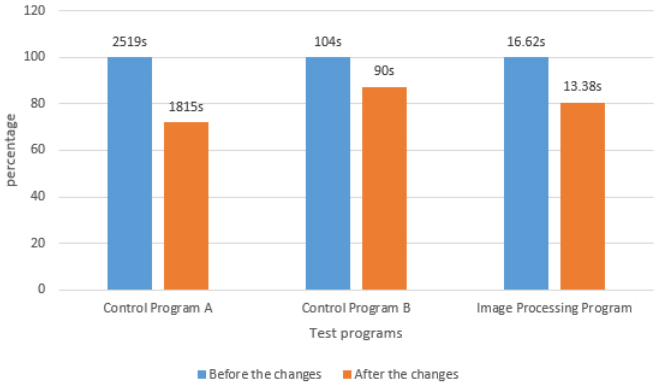
Mehta et al. [1] tried to reduce the number of valid statements by combining consecutive multiple statements and their data and control dependency as a super statement. Among these super statements, the statement condensation is executed to combine the areas of the same iteration together. In most of the cases, because the dependence distance among the statements in the large scale programs is very short and close to the shape of dependent vector, it is possible to reduce the amount of dependencies needed to be focused by applying statement condensation.

Nick et al. proposed another approach to this topic by mainly generating the dependence graph in different way [2]. When the compiler is exporting the highly analysis to the program, it is common to use Directed Acyclic Graph of Strongly Connected Components (DAGscc) based on the Program Dependence Graph (PDG). The amount of information stored in DAGscc is much smaller than PDG. Therefore, there is an opportunity to reduce the computation cost. However, it is general to make DAGscc after the complete PDG is generated. To reduce the computation time of making DAGscc, they tried to eliminate the redundant value of no affecting to the DAGscc while the PDG is being generated. At the same time, because the analysis is implemented with Demand-Driven, only the necessary parts but not the whole source program will be analyzed.

Besides the researches above, a fine-grain redundancy removal precompilation technique for accelerating the compilation time of large C/C++ programs was proposed by Yu et al. [14]. This technique aims for removing the multiple



(a) Time reduction in analysis



(b) Time reduction in total

Figure 5. Time reduction of the program analysis passes for evaluated programs. Basic analysis includes def-use analysis and dependence calculation. Each bar shows the normalized analysis time for each analysis group based on OSCAR compiler without the proposed technique.

falsely declaration in the header files when they are used in different compilation units (e.g. '.c file') but not all of the declarations are useful. Although the functionality of a system is not affected, redundancies and false dependencies in the precompilation units affect the efficiency of the development process and the total build time. This technique extract the compilation units firstly into a sequence of program units (PU), which means the declaration of each variable, then remove the redundancy based on the abstract syntax tree. Finally, partition and regroup the necessary PU's into the header and compilation units to reduce the unnecessary header files.

Different from these previous works, our technique focuses on the organization of the parallelizing compilers, especially multiple program analysis passes and restructuring passes. Furthermore, our technique takes care of the global structure of a source program such as the caller-callee relationship and the global effects by global scope objects.

7. Conclusion

In this paper, we address the key compilation time problem at the powerful optimization especially for parallelizing compilers, namely OSCAR compiler, which usually apply aggressive program restructuring to the target program. However some of the functions may remain the structure before the program is restructured because a restructuring pass does not always affect the whole part of the program. We therefor proposed the technique of removing redundant program analysis considering inter-procedural propagation of analysis update information. In this technique, the compiler records the functions whose structure is changed after restructuring the program, then the compiler analysis the functions recorded to update the information of each analysis. For the remained functions whose structure are not affected by the restructuring, the previous analysis information can be used again.

The proposed technique is employed in OSCAR automatically mutltigrain parallelizing compiler. We evaluate our proposed technique in three proprietary large scale real applications. For image processing, the time cost in the pointer analysis and the basic analysis are reduced by 43% and 35% separately. For Control Program B, they are 59% and 41%, and finally for the Control Program A, they are 53% and 37%.

Acknowledgments

This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] S. Mehta and P-C. Yew. Improving Compiler Scalability: Optimizing Large Programs at Small Price. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), pp. 143–152, 2015.

[2] N. P. Johnson, T. Oh, A. Zaks and D. I. August. Fast Condensation of the Program Dependence Graph. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13), pp. 39–50, 2013.

[3] U. K. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[4] K. Ishizaka, T. Miyamoto, J. Shirako, M. Obata, K. Kimura and H. Kasahara. Performance of OSCAR Multigrain Parallelizing Compiler on SMP Servers. Language and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, west Lafayette, USA, pp 319–331, 2005.

[5] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka and H. Kasahara. Hierarchical Parallelism Control for Multigrain Parallel Processing. Language and Compilers for High Performance Computing, 15th International Workshop, LCPC 2002, college Park, USA, pp 31–44, 2005.

[6] Y. Peak, J. Hoeflinger and D. Padua. Efficient and Precise Array Access Analysis. ACM Transactions on Programming

Languages and Systems (TOPLAS), Vol. 24, Issue 1, pp. 65–109, January 2002.

[7] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara and S. Narita. A multi-grain parallizing compilation scheme on OSCAR (optimally scheduled advanced multiprocessor). Proc. 4th Workshop on Language and Compilers for Parallel Computing (LCPC91), pp. 283–297, 1992.

[8] E. M. Nystrom, H.-S. Kim and W. W. Hwu. Importance of heap specialization in pointer analysis. Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2004), pp. 43–48, 2004.

[9] M. Emami, R. Ghiya and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94), pp. 242–256, June 1994.

[10] M. Hind, M. Burke, P. Carini and J.-D. Choi. Interprocedural Pointer Alias Analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 21 Issue 4, pp. 848–894, July 1999

[11] J. Shirako, K. Nagasawa, K. Ishizaka, M. Obata, H. Kasahara. Selective Inline Expansion for Improvement of Multi Grain Parallelism. The IASTED International Conference on PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS, 2004.

[12] D. Umeda, T. Suzuki, H. Mikami, K. Kimura and H. Kasahara. Multigrain Parallelization for Model-based Design Applications Using the OSCAR Compiler. Language and Compilers for Parallel Computing (LCPC2015), pp. 125–139, 2016

[14] 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), pp. 59–68, September 2005.

[14] 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), pp. 59–68, September 2005.