A SIMPLE DATA DEPENDENCY ANALYZER FOR C PROGRAMS

Gian Chand Sodhy & R.K. Subramanian
School of Computer Science
Universiti Sains Malaysia
11800 Pulau Pinang
Malaysia

Abstract

Data dependencies that exist in a sequential program are a major hindrance towards parallelization. Dependencies between simple statements are relatively easy to detect. However, there are a lot of dependencies that are hidden due to procedure calls. Most of interprocedural data dependencies are caused by the use of global variables, passing of information via parameters and also by aliasing. Furthermore, data dependency analysis is a complex task, requiring much analysis and resources. Our work reports on the development of a simple interprocedural data dependency analyzer for C programs incorporating simple techniques. A read and write set is created for each statement, which are then compared for overlapping variable names. Using this technique, we are able to check dependencies between statements, across procedures, within a procedure, array loops, IF-THEN-ELSE blocks as well as standard library function calls. Results from this analyzer is a dependency table which can be used by a parallelizing compiler.

Keywords

Parallelizing compilers, Interprocedural analysis, Data dependencies

1.0 Introduction

Dependency analysis is one of the major tasks of a parallelizing compiler. Basically it concerns checking statements in a program for dependencies. If two statements are dependent, they must be executed in the order specified in the source code. Otherwise, they can be run in parallel.

Data dependencies are caused by the flow of data in a program. If a statement refers to a variable whose value is calculated in another statement, the two statements are said to be dependent. This implies that these two statements cannot be parallelized because one statement depends on the other.

2.0 Types of dependencies

There are two types of dependencies that could exist in a program, namely data dependency and control dependency. Data dependency is caused by the usage of data whereas control dependency is due to flow of control in the program.

Data dependency can be further divided into three types - flow, anti and output. Assume S1 and S2 are two statements in a sequential program with S2 following S1. If S1 computes a value that is subsequently used by S2, then S2 is flow-dependent on S1. S2 is anti-dependent on S1 if S1 uses a value (of a variable) that is subsequently recomputed by S2. If both S1 and S2 compute the same variable, then S2 is output-dependent

on S1. Formal definition of these dependencies has an extra condition: there should be no intermediate statement between S1 and S2 that changes the value of the variable which causes S2 to be dependent on S1. This makes the dependency computation more complex and expensive to implement.

Control dependency occurs when the execution of one statement depends on another. It is normally present in IF-THEN-ELSE statements. Statements in THEN section are only executed if the condition in IF is satisfied, otherwise statements in ELSE section will be executed. This makes statements in THEN and ELSE section to be control dependent on the statement containing the IF condition.

Dependencies can also exist in loops, which is a major source of parallelization. If a variable's value is computed and used in the same iteration of the loop, the dependency that exists is loop-independent (as the value of the variable is not needed in the next iteration). On the other hand, when a value is computed in one iteration but used in a different iteration (or vice-versa), a loop dependency exists between the statements concerned. Loops that do not contain dependencies between iterations can be parallelized easily by assigning each iteration of the loop to a separate processor. This cannot be done for loops containing dependencies between iterations, requiring some synchronization between the processors.

For certain types of dependencies, the parallelizing compiler will be able to remove or reduce them by reordering or modifying the statements concerned (while still guaranteeing correctness).

Dependency testing has two goals. One, it tries to disprove dependency between pairs of references. If dependencies exist, it tries to characterize them in some manner. Dependency testing must also be conservative, and assume the existence of any dependency it cannot disprove. Otherwise the validity of any parallelization based on dependency information is not guaranteed.

While data dependency caused by scalar variables are straightforward, those caused by array references are complex. In the case of array variables, it may be difficult to determine when two subscripted array references refer to the same element of the array. Hence, some specialized tests to detect dependencies in arrays have been reported. They include greatest common divisor (GCD) and Hierarchical Dependency Framework using Banerjee-Wolfe decision algorithm [1], distance and direction vectors [2], dependency tests based on Banerjee's inequalities [3], Omega test [4], range test [5], and many others.

2.1 Interprocedural dependencies

There is another type of dependency which hinders parallelization - interprocedural. This happens when a statement in one procedure accesses the same memory location as another statement from a different procedure (or from the main program). The variables used in the related statements might be different but if the same memory location is accessed, dependency occurs. Obviously this type of dependency is more difficult to detect and requires more detailed analysis.

More often than not, statements involving procedure calls are ignored and interprocedural dependency not determined. This could be a major source of parallelization. A whole procedure could be run in parallel if it is independent, saving considerable execution time. However, a compiler might take the conservative approach, and assume dependencies for all parameters passed into the procedure. This doesn't help much in terms of parallelization.

2.2 Sources of interprocedural dependencies

There are a number of possible causes for dependencies to exist among procedures [6], mainly involving global variables, parameters and aliases. Whenever a global variable is used by a statement and also appears in a procedure, there exists a data dependency between the procedure and the statement (outside of the procedure).

Parameters are used in procedures to pass information either by value or by reference. Passing by value involves copying the content (or value) of the variable which is being passed as a parameter. This ensures that the contents are only read in the procedure, and cannot be modified. Pass by reference refers to passing the address of the variable used as a parameter. Here, the contents of the variable can be modified in the procedure. Therefore parameter passing can cause dependencies, although it is not so obvious at first glance (due to different names used for the variables passed and parameters concerned).

Aliasing occurs when two or more objects having different names refer to the same memory location. In the context of a procedure, a local variable could be assigned to the same memory location as a global variable

Various techniques have been proposed to perform interprocedural analysis on programs. One of these is in-line expansion. It expands procedure calls by substituting each call statement with the body of the called procedure. After expansion, the entire program consists of only one procedure [7]. Appropriate substitutions have to be made so that the expanded version is semantically identical to the original code. By eliminating the procedure call, the need for analysis of the procedure is eliminated too. However, this method increases program size since each call to a procedure is replaced with a separate copy of the procedure body. Furthermore, it can't handle recursive calls. In-line expansion abolishes program modularity and the resulting code becomes less readable and more difficult to maintain. Other methods basically deal with array references in loops from different procedures, such as linearization [1], atoms and atom images [7] and global regions [8].

Most of these techniques are quite complex and require detailed analysis and heavy processing. For our purposes, we have used very simple techniques which do not require heavy processing, but nevertheless produces sufficient results. The design of our analyzer is described next.

3.0 Our simple data dependency analyzer

We have developed a tool [9] that can analyze a sequential C program for data dependencies. It generates a data dependency table containing information related to the dependencies detected. The contents of this table can be used by a parallelizing compiler for further processing.

3.1 Methodology

A simple method to detect data dependencies between statements is by using read-write sets. For each statement, a read and a write set is created. A read set consists of all variables that are referred (or being read) in a statement. A write set contains variable(s) that are modified (or written) in a statement. In an assignment statement, a variable on the left of the assignment symbol (i.e. '=') is written to, whereas all variables appearing on the right hand side of the assignment symbol are being read. By constructing and comparing read and write sets for all statements, different types of data dependencies can be determined. If a variable happens to appear in two statements and one of them is a write, then

we can say that the two statements are dependent (in terms of that variable).

The read set of a statement is compared, one by one, with all the write sets of statements above. If these sets overlap, it indicates one or more variable names are present in them. The overlapping variable names are extracted and stored as individual entries in the dependency table. Since a variable from a read set also appears in a write set above it, this means that the variable concerned was written to before it was read. This is a case of flow dependency and recorded as such (for the variable concerned) in the dependency table.

Next, the write set of the same statement is compared with all read sets above it. Any overlapping between the compared sets indicates the presence of common variable(s) that are read before they are written to. This shows the existence of an anti dependency with regard to the two statements.

Another comparison is made between the write set of the statement and all the write sets above it. If they overlap with regard to some common variable, it means the variables concerned are written to in both the statements. An output dependency exists between the compared statements in terms of the shared variables.

In short, when a variable occurs in two statements S1 and S2, where S1 is followed by S2 (in order of execution):

- flow dependency exists between S1 and S2 if the variable is present in write set of S1 and read set of S2;
- anti dependency exists between S1 and S2 if the variable is present in read set of S1 and write set of S2;
- output dependency exists between S1 and S2 if the variable is present in write sets of both S1 and S2;

plus the variable is not written to (i.e. do not appear in write set) in any of the intermediate statements between S1 and S2.

The location of a statement also determines which other statements it should be compared with (in terms of dependencies).

Analysis is performed in two passes. In the first pass, global variables, constants and user-defined procedures are recognized, extracted and stored. For each procedure, a summary is produced containing its name, list of formal parameters, parameters read/modified and global variables read/modified. The second pass analyzes statements in the main program, using the information that has already been accumulated from the first pass. Statements are checked for variables that are written and read. When procedure calls are encountered, the effects of the procedure call (in terms of variables passed in as parameters as well as global variables) are determined from the information stored during the first pass. This enables "hidden" actions performed in a procedure on variables passed as parameters as well as global variables to become known to the dependency analyzer.

Pointers and aliases are identified during both these passes - from the procedure body in the first pass, and from the main program in the second pass.

For FOR loops, its header is analyzed to extract the loop index variable. Array references in a FOR loop are treated differently. If the array reference contains the loop index variable, then it is possible for loop-carried dependency to exist. Hence the whole array reference is kept in the read or write set. But, on the other hand, when an array reference does not refer to the loop index variable, this means that the array variable cannot be loop-dependent. Therefore, only the array name is kept in the read or write set of that statement.

Pointer analysis includes recognition of global or local types, its location as well as pointer-type (formal) parameters in the procedure header. Local pointers can be either in the main program or in some procedure. We do not consider FILE pointers, since they are basically handles

to files for reading and writing purposes, and not used in the same way as normal pointer variables.

In a C program, an alias occurs when a pointer is assigned to another pointer, or a pointer is assigned the address of a variable (indicated by preceding the variable name with '&'). Furthermore, it must be a direct assignment (without any other variables in that statement). This is known as an alias-pair. Both the names in the alias-pair are aliases to each other. Using either of them will involve the same memory location. Statements containing different variables appear to be non-dependent at first glance, but if any of the variables have aliases, this information must be utilized to find "hidden" dependencies. Our analyzer works in the following manner. When the read-write sets are created for a statement, the variables concerned are checked (one by one) if any of them are pointers. For aliasing to occur in a statement, the variable being written (on the left-side) must be a pointer, whereas the variable being read (the right-side) is either a pointer or the address of a variable. We only identify alias pairs and record them, but do not include them for computing dependencies. Alias analysis is a complex and intricate subject, thus requiring detailed understanding and analysis [10,11].

3.2 Dependency checking

Dependencies between statements are checked using the read/write sets. Later, this can be used to determine whether or not a section of code can be parallelized.

Statements appearing in IF-THEN-ELSE blocks are treated differently when checking for dependencies. Dependencies are not checked between the two sets of statements that are controlled by IF-THEN and ELSE (even though one or more variables appear in both of them). However, statements appearing in the same section of the IF-THEN-ELSE block are checked for dependencies since they are executed in sequence. A statement appearing outside of this block is compared with all the statements in it (in both sections of IF-THEN and ELSE).

Local variables of a procedure, do not interfere with variables (even of the same name) from other procedures. Hence, statements in a procedure are treated as a block, and dependencies are checked among them. Statements containing references to parameters and global variables are checked for dependencies in the same way.

When a procedure-call is encountered in a statement, the procedure's summary information is used. Actual parameters (from the calling program) are mapped onto the formal parameters of the procedure. A new set of variables (of the calling program) being written in a procedure is created by combining the procedure's sets of written parameters and global variables. Likewise, a new set of variables read in the procedure is created by combining the procedure's set of parameters read and set of global variables read. This new set of read and write variables in the calling program captures the essence of calling the procedure. In other words, the read and write sets of the statement containing a procedure call incorporates the effects of running the procedure, in terms of parameters passed and global variables.

Now, the read and write sets of this statement (with a procedure call) can be compared with related sets from other statements for any dependencies. With this, the interprocedural effects on variables passed as parameters and global variables are made known. By checking statements in the main program (using the read-write sets), dependencies caused by the execution of a procedure can be detected.

Calls to standard library-function can also cause dependencies, in terms of parameters used. Since the source-code of library functions is not available, we have

decided to transcribe C library functions in terms of parameters written and read during their execution. Whenever a library function call is encountered, a its read and write sets are obtained and used to determine dependencies by comparing it with similar sets of other statements in the program.

4.0 Experiments

In order to test our analyzer, we used a number of different real-life C programs. Each of these C programs was analyzed one at a time. Although the programs chosen for tests were not overly complex in terms of code (e.g. no nested IFs and loops), nevertheless they contained most of the essential elements such as procedures, global variables, constants and pointers.

The results obtained of these tests were satisfactory, as dependencies were detected correctly in most cases. However, some complex cases of loop-related dependencies in arrays were not detected due to insufficient analysis involving array elements and loop index variables. As mentioned earlier, dependencies caused by aliases were also not considered.

5.0 Conclusion and future work

Data dependencies that exist in a sequential program are a major hindrance towards parallelism. They have to be detected and made known to a parallelizing compiler, which will try to generate efficient parallel code while still guaranteeing correct results.

Being one of the major components of a parallelizing compiler, data dependency analysis has to be accurate. All possible dependencies that are present must be detected. Dependencies between simple statements (without procedure calls) is relatively easy to detect. However, there are a lot of dependencies which lie hidden due to procedure calls. The very purpose of using procedures is to hide details from the main program in order to produce more programmer-friendly code which is also modular. But this same fact makes dependency checking a more difficult task. Furthermore, new dependencies are introduced during procedure calls, some of which the programmer may not even be aware of.

This clearly shows the need for an interprocedural data dependency analyzer, which will be able to find "hidden" dependencies. Most of these dependencies are caused by the use of global variables, passing of information via parameters and also by aliasing.

We have shown how a simple data dependency analyzer can be built using very simple techniques. Our analyzer manages to detect dependencies caused by program flow (i.e. flow, anti and output) as well as procedure calls. On top of that, we have also incorporated a simple method to detect possible dependencies due to use of standard library functions.

One of the important results from this analyzer is the dependency table. It captures all data dependencies that exist in the C program (i.e. dependent variables, types of dependencies, its location, etc.). The dependencies are organized according to procedures. Dependencies in the main program are the most important as they indicate whether or not a statement is dependent on other statements. This allows the parallelizing compiler to consider parallelizing statements (which might include a procedure call). If a statement containing a procedure call is independent, it means that the procedure can be executed in parallel. On the other hand, if the procedure call is dependent on some other statement, then the parallelizing compiler can analyze the individual statements in that procedure to see if some of them can be parallelized instead.

For future work, the analysis and detection can be further improved by making use of more sophisticated techniques. Among improvements to be considered are incorporating detection of

other types of dependencies (e.g. control, cyclic, reduction/induction), eliminating some cases of false dependencies [12], and a more detailed alias analysis (for both static and dynamic types) to improve dependency checking.

6.0 Acknowledgement

We gratefully acknowledge the research grant provided by Universiti Sains Malaysia, Penang that has resulted in this article.

7.0 References

[1] Michael Burke and Ron Crytron, "Interprocedural Dependence Analysis and Parallelization," in *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction, ACM SIGPLAN Notices,* Vol. 21, No. 7, Palo Alto, California, June 25-27, 1986.

[2] Michael J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1982.

[3] Utpal Banerjee, *Dependence Analysis of Supercomputing*, Kluwer Academic Publishers, 1988.

[4] William Pugh, "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis," *Communications of the ACM*, 8:102-114, August 1992.

[5] William Blume and Rudolf Eigenmann, "The Range Test: A Dependence Test for Symbolic, Non-linear Expressions," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Technical Report 1345, April 1994.

[6] Gian Chand Sodhy and R.K. Subramanian, "Interprocedural Data Dependency Analyzer for C Program," in *Proceedings of the IASTED International Conference - Applied Informatics* (M.H.Hamza, editor), Innsbruck, Austria, February 19-22, 2001, pp. 350-354.

[7] Zhiyual Li and Pen-Chung Yew, "Efficient Interprocedural Analysis for Program Parallelization and Restructuring," *SIGPLAN NOTICES*, Vol. 23, No. 9-12, 1988.

[8] R. Triolet, F. Irigon and P. Feautrier, "Direct Parallelization of Call Statements," *ACM SIGPLAN'86 Symposium on Compiler Construction*, 1986, pp. 176-185.

[9] Gian Chand Sodhy, "Interprocedural Data Dependency Analyzer for C programs," M.Sc. thesis, Universiti Sains Malaysia, March 1999.

[10] R.Parimaladevi, "Developing a Software Tool to Identify and Eliminate Aliases in C programs for Data Dependency Analysis," M.Sc. thesis, Universiti Sains Malaysia, October 1997.

[11] R.Parimaladevi and R.K. Subramanian, "Aliases and their Effect on Data Dependency Analysis," *Malaysian Journal of Computer Science*, Vol. 10, No. 2, December 1997.

[12] William Pugh and David Wonnacott, "Static Analysis of Upper and Lower Bounds on Dependences and Parallelism,", Dept. of Computer Science, University of Maryland, College Park, Technical Report CS-TR-3250, March 1994.