Testing Method of Code Redundancy Simplification Based on Program Dependency Graph

Yang Fan
Computer School
Wuhan University;
China University of Geosciences
Wuhan, China
planesail@163.com

Zhang Huanguo
Computer School
Wuhan University
Wuhan, China
liss@whu.edu.cn

Yan Fei
Computer School
Wuhan University
Wuhan, China
3641776@qq.com

Yang Jian
Computer School
Wuhan University
Wuhan, China
dayangyou@163.com

***Abstract*—This paper presents a testing method of code redundancy simplification from the aspect of software static analysis. We study on the dependency relationships between the variables, branches and functions in source or intermediate code of the tested object by PDG (Program Dependence Graph). This method establishes an effective testing to discover and locate the redundant functional modules and the unreachable paths based on dependency relationship. Compared with the conventional code optimization which emphasizes the improvement of time efficiency, we compress the size of source code and object code, independent of the programming language the source program uses. Experimental results show that the source or object code size decreases by 1 to 3 percent approximately after our simplification, not only reducing the occupied space but also assuring the functional consistency.**

Keywords-Code Redundancy Simplification; Static Analysis; Program Dependency Graph; Unreachable Path

I. INTRODUCTION

The most commonly used ways of software testing include validity testing and functionality testing. In addition to the conventional test items, the code size of software is an important testing index as well [1]. In some specific application, such as embedded system with limited computing resources, it is of great significance to decrease the size of code occupancy as far as possible and guarantee the consistency of code functionality in the meantime for both functionality and performance.

Software implementation should take many aspects into consideration, such as code amount, code clearness, code validity, code stability and code reliability [2]. Redundancy is one of the primary technologies for system fault tolerance, which efficiently improves the informational reliability, availability and security. Given other conditions constant, rational redundancy can enhance software credibility. However, superfluous redundant code not only occupies more storage space but also introduces more probable security faults, leading to the increase of fault rate and the decrease of overall security [3]. The Department of Homeland Security has uncovered an average of one security glitch per 1000 lines of code in 180 widely used open-source software projects [4]. In order to reduce the size of redundant code, we should discover the redundancy of variables, branches and functions, and locate the code corresponding to superfluous redundant functionality and unreachable paths. Currently, code redundancy simplification needs further and deeper study.

To evaluate the code size accomplishing the prescribed functionality precisely, it is not only a core problem of software development and testing, but also a universal problem of basic computer science. On theory, it's an extremely tough scientific problem.

Our work is a special testing method aiming at the redundancy simplification of software code. First, our testing is based on static analysis without the actual program running so as to analyze globally and cut down the workload during the program's execution. Second, we use PDG to illustrate and analyze software, which discovers and locates the redundant data, branches and functional modules vividly and effectively. Third, our method is appropriate for both source code and object code at two different levels, unlimited of which language the source program uses or whether we have the source code.

The remainder of this paper is organized as follows. Section 2 reviews previous related work. Section 3 introduces the PDG from the view of software static analysis. Section 4 describes our testing method of code redundancy simplification based on PDG. Experiment results and evaluation analysis are presented in Section 5. Section 6 concludes this paper and discusses our limitations.

II. RELATED WORK

Code minimization of the prescribed functionality is a NP-Complete problem and there's no relevant research results published up to present. Relatively speaking, code optimization is already a mature research topic.

Hennessy, J introduced a heuristic algorithm to rearrange the instructions at compile time to avoid pipeline interlocks of VLSI [5]. Jens Knoop presented the lazy code motion, minimizing the number of computations of program while suppressing any unnecessary code motion to avoid superfluous register pressure [6]. Keith D. Cooper employed a genetic algorithm to find optimization sequences that generate smaller object codes during compiling [7]. Santosh Chede used the constrained optimization method to obtain

978-0-7695-4745-9/12 $26.00 © 2012 IEEE
DOI 10.1109/TrustCom.2012.269

the optimized code size, execution time, energy consumption in the real-time embedded system [8]. Zhang Dian applied intermediate code optimization based on GCC compiler and data flow analysis [9]. T. Simunic adopted code optimizations of software performance and energy consumption at three levels of abstraction: algorithm, data and instruction-level in embedded system [10]. E. Ozcan proposed a Memetic Algorithm (MA) to arrange the best number of processors and the best data distribution for each stage of a parallel system [11]. R. Kirner studied the Worst-case execution time analysis, particularly emphasizing on execution time [12]. The existing methods of code optimization focus on the compiling process predominantly, possessing low generality. Some optimizations of embedded software rely on the specific hardware platform and compiler environment. The usual methods aim at the improvement of time efficiency, not the decrease of code size.

Program Dependence Graph (PDG) provides an intermediate program representation, widely used in generating reduced test sets, vectorization, parallelization, regression testing, static and dynamic slicing.

ALLEN, J originated a method of systematically converting control dependence to data dependence, revealing the relevance between them [13]. J. Ferrante presented an optimization way [14], by which a single traverse of these dependences was sufficient to perform several optimizations in a manner uniform for both control and data dependences. R. A. Ballance presented new methods of constructing control dependence graph and data dependence graph without the traditional use of control flow graph [15]. M. J. Harrold constructed PDG during the parsing phase of program, obtaining the intermediate presentation from source code [16]. Pinter, S demonstrated a translation procedure for making programs run efficiently on parallel machines by PDG representation [17]. Binkley, D proposed results improving the performance for graph-based interprocedural slicing on System Dependence Graph (SDG) [18]. C. Hammer explained PDGs for sequential and multi-threaded programs, and showed precision gains due to flow-, context-, and object-sensitivity [19]. M. Harman introduced three algorithms which reduced the number of tests without compromising coverage achieved by using control dependence graph [20]. The usual applications of PDG include the parallelization of programs, software slicing and test data reduction, the latter two of which give us clues of code redundancy simplification.

III. PROGRAM DEPENDENCE GRAPH

A. Definition of PDG

A Program dependence graph (PDG) is a directed graph, G = (V, E), where the vertices V are operators, statements, regions and predicate expressions, and the edges E, represent both data and control dependence on which values and operations at a vertex depend.

PDG is considered as a combination of two different-layer subgraphs: a data dependence subgraph (DDS) and a control dependence subgraph (CDS). Edges denoting data dependences and the corresponding vertices (sources and targets) form the DDS. Edges denoting control dependences and the corresponding vertices (sources only) form the CDS.

Definition 1 Let P=<V, E> be a program dependence graph. The *data dependence subgraph* of P is the subgraph DDS_P=<V', E'> where E' = {e E: e represents a data dependence} and V' = {v, v' V: <v, v'> E'}.

Definition 2 Let P=<V, E> be a program dependence graph. The *control dependence subgraph* of P is the subgraph CDS_P=<V', E'> where E' = {e E: e represents a control dependence} and V' = {v V: <v, v'> E'} ∪ {Entry}, in which Entry is the only root vertex of PDG.

B. Control Dependence

Control dependence formalizes the notion that execution of one vertex conditionally depends on the execution of another vertex in CDS. Control dependence is formally defined in accordance with the postdominance relationship on the control flow graph.

Definition 3 A *control flow graph* (CFG) is a directed graph whose vertices are the basic blocks of a program together with two distinguished vertices Entry and Exit. Every vertex in the graph has two successors at most with the attributes T (true) and F (false). There exists an edge from Entry to a basic block demonstrated by a vertex and an edge from a corresponding vertex to Exit as well.

Definition 4 Assume X and Y are vertices in the CFG. Y *postdominates* X (Y is postdominated by X) iff Y appears on every path from X to Exit.

Definition 5 Let G be a control flow graph. Let X and Y be vertices in G. Y is *control dependent* on X iff
(1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and (2) X is not post-dominated by Y. If Y is control dependent on X, X must have two exits. Traversing one of the exits from X always results in Y being executed, while following the other exit Y will not be passed through.

A CDS contains several types of vertices. Statement vertex, illustrated as ellipse, represents statements in the program. Region vertex, illustrated as circle, summarizes the common control dependence relationship for statements in the region. Predicate vertex, illustrated as square, produces two edges with boolean value True and False. Figure 1 demonstrates a program segment with its corresponding CFG and CDS. For example, we can infer from Figure 1 that S5 is control dependent on P1-false, independent of P2's value.

Figure 1. Control Flow Graph and Control Dependence Subgraph

C. Data Dependence

The DDS contains three types of data dependences, representing flow-, anti- and output-dependence respectively.

Definition 6 Assuming statements S1 and S2, S2 is *data dependent* on S1 if: [I(S1) ∩ O(S2)] ∪ [O(S1) ∩ I(S2)] ∪ [O(S1) ∩ O(S2)] ≠ Ø where (1) I(Si) is the set of memory locations read by Si and (2) O(Sj) is the set of memory locations written by Sj and (3) there is a valid execution path from S1 to S2.

There exist three cases:

- *Flow-dependence*: O (S1) ∩ I (S2), S1 → S2 and S1 writes something before S2 reads it.
- *Anti-dependence*: I (S1) ∩ O (S2), S1 → S2 and S1 reads something before S2 overwrites it.
- *Output-dependence*: O (S1) ∩ O (S2), S1 → S2 and both write the same memory location.

The corresponding data dependences will be interpreted in the following sections.

IV. Testing Method of Code Redundancy Simplification

Code minimization accomplishing the prescribed functionality hardly has the universal model and method, which is a NPC problem in the view of computational complexity. However, we can reduce the code size for some specific software to some extent. For problem simplification, we set about studying on the testing method of code redundancy simplification. Testing on code redundancy simplification of accomplishing the prescribed functionality refers to the testing approach of decreasing the code size as far as possible, under the premise of guaranteeing the consistency and security of the original code. We emphasize not the proving and achievement of the code minimization on theory, but the discovery and location of the superfluous redundant functional modules and the unreachable paths. In this way, we can obtain the advice and guidance for code simplification, and consider the code redundancy simplification as satiable or feasible.

A. Generation of PDG based on source and object code

Our testing method of code redundancy simplification is appropriate for both source code and object code at two different levels comprehensively. Therefore, we convert source code and object code into PDG form firstly.

For source code, we adopt M.J. Harrold's algorithm ConstructCDS [16] using the concept of "condition stack", where input is an abstract syntax tree (AST) for a procedure P and output is the CDS for P. AST is a tree representation of the abstract syntactic structure of source code, but no limited to what language the source code uses. We obtain AST by TREEGEN algorithm [21], employing seven generation rules with the input of standard BNF (Backus-Naur Form) representation. The details of ConstructCDS and TREEGEN can refer to the relevant references.

For object code, general algorithms for computing control dependence information construct both the reverse of the control flow graph and its post-dominator tree, applicable for the situation without source code. The post-dominator tree encodes the global program flow information needed to compute a CDS [14]. For any executable file, we can disassemble the object code to gain control flow graph by IDA (a static disassembling tool) [22].

Then, we use Tarjan's interval analysis [23] algorithm to calculate the data dependence relationship from CDS. In Tarjan's method, an INTERVAL refers to a strongly connected component in CFG. To interpret this algorithm, we define the notion of REACHUNDER at first.

Definition 7 Let CFG=<V, E> be a control flow graph and x be a vertex that is the target of a back edge in the CFG. Then the interval with the header vertex x is defined by the set *REACHUNDER(x)* = {x} ∪ {y∈V so that y is not yet a member of any interval and along a path y can reach x but not passing through x whose final edge is a back edge}.

The Tarjan's interval analysis algorithm is as follows [15]:

- Generate a depth-first spanning tree (DFST) of the CFG. Number the vertices sequentially based on their appearance in a preorder traversal. Mark back edges in the DFST.
- For each back edge <v, x> compute REACHUNDER(x). Back edges are processed in reverse of their preorder appearance to assure that intervals will be detected from the inside out. Replace the vertices in REACHUNDER(x) with a new virtual vertex representing the interval whose elements are the vertices in REACHUNDER(x).
- If it traverses a vertex y that is not a descendant of x in the DFST, the graph is not reducible. The method of Schwartz and Sharir can be used to isolate the irreducibility within an improper interval [24].
- Once all back edges have been processed, the remaining vertices and intervals can be aggregated into an outermost interval.

After getting both the CDS and DDS, the core steps of redundancy simplification on PDG begin.

B. Node Merging

PDG avoids unnecessary ordering in the original source code or control flow graph, representing program in parallel relationship.

In CDS, we scan all region nodes, which summarize the common control dependencies for statements. We can merge several region nodes and their region domains given (1) the conditional clauses of different region nodes are the same or have the same values, (2) statements in the region domains are data independent without dependence edges in the DDS. After merging, redundant branches and calculations of conditional clauses are removed from CDS, helpful to generate simplified object code.

C. Loop Fusion

Loop fusion refers to joint or fuse several loop bodies to form a single loop to decrease the code size.

Loop fusion should meet the following three conditions: (1) the loops are executed under exactly the same control conditions, (2) there is no data dependence between the two loops, and (3) the loops are executed the same number of times. Similar to node merging idea, the object of loop fusion is the loop body, while the node merging fuses the descendent dependence branches of conditional predicates.

D. Branch Deletion

The control dependence update in the PDG is easily performed when a branch is deleted. Suppose a predicate node P is found to be always true. If there is a P-false region node, it is pruned along with all of its control dependence successors, called the *unreachable path*. If there is a P-true region node, all successors of the region node are made successors of P's control dependence predecessor and then both P and the region node will be deleted. However, the deletion of a conditional branch should take the data dependence into consideration as well. If the involved statements include data dependence of the other branch or the following part, then the deletion operation shouldn't be taken.

E. Generation of simplified code

Simplification of PDG is the optimization of intermediate code. For the practical application, we should convert the intermediate presentation into object file at the last step.

After the redundancy simplification of our testing method, the simplified code is obtained from source code or object code. Final reduced executable file is generated by common compiling based on PDG graph form, such as GNU C compiler GCC, which is suitable for PDG.

V. Experiments

In this section, we conduct two experiments. The former shows one example of redundancy simplification for a specific code fragment based on its PDG. The latter displays two groups of comparison results on both the source code and object code.

Our experimental hardware platform is PC of Intel Pentium 4 2.8GHz CPU and the Operation System is Ubuntu 7.01. We use GCC 3.4 for compiler and IDA 5.2.0 for static disassembler.

In Experiment 1, we illustrate the simplification application of both node merging and branch deletion on a specific example PDG.

Figure 2 shows a code fragment and its corresponding PDG, where the left is the pseudo-code of source or intermediate code and the right PDG includes both CDS and DDS. In PDG, the dashed lines indicate the edges of CDS and the solid lines indicate the edges of DDS. Line 3 is output dependent on line 1 because both 3 and 1 write the same memory unit A, while the other data dependences are flow dependences.

Figure 2. Code Fragment and its PDG

During the scanning procedure, the predicates of P1 and P2 are consistent on the "true" or "false" value. Given C and D are not data dependent on A and B, we can reduce the code size by node merging of P1 and P2 so that line 12, 14 and 16 are merged into line 2, 6 and 9, and line 5 and line 8 can also be pruned for the repetition assignment of boolean value of True and False. By node merging, the optimization rate is 31.3%.

Given the predicate of P1 is always true, we can simplify this code by branch deletion. According to the right PDG, the false branch of predicate P1 is unreachable path, this branch and P1 itself can be deleted, i.e. line 2, 6, 7, 8 and 9. As line 3 and 1 has the data output dependence, where the value of A has assignment at line 3 again, A is no more dependent on line 1. As a result, line 1, dependence edge (1, 3) and edge (1, 10) can be deleted. By simplification, the line 1, 2, 6, 7, 8 and 9 of left code can be pruned, with the optimization rate of 37.5%.

When combining this two simplification methods, we get the result code fragment and the simplified PDG in Figure 3, where the optimization rate is 68.8%. The result is very ideal for this specific example.

Figure 3. Simplified code fragment and its PDG after node merging and branch deletion

In Experiment 2, we compare the simplification effects on the corresponding groups of source code and object code. We list two groups of source and object code and their respective simplified results in Table 1. The testing objects are two Unix utilities: sort and tail.

TABLE I. Simplification of Source and Object Code

Program	Program Size of PDG nodes before Redundancy Simplification	Program Size of PDG nodes after Redundancy Simplification	Optimization Rate
sort (source)	6,377	6,328	0.77%
sort (object)	5,820	5,739	1.39%
tail (source)	2,831	2,763	2.40%
tail (object)	2,580	2,503	2.99%

As the PDG is the object of redundancy simplification, we take the node number of PDG as the measurement of the optimization effect. According to our preliminary experiment, we can find the optimization rate of source code is a little lower than object code. It is inferred that the intermediate form of the object code by disassembling is more fine-grained than the AST translation of the source code, resulting in the higher simplification rate.

Our experiments condense the code size of source code and object code, preserving the functional consistency.

VI. Conclusions

Static analysis on program can list all static behaviors exhaustedly, independent of specific running environment. PDG combine both data dependence and control dependence, avoiding the unnecessary ordering in the control flow graph and source code. As the dependence in PDG connect the corresponding parts in the calculation of program, one traversal of PDG can carry out several simplification operations on both CDS and DDS with high efficiency. Code redundancy simplification is appropriate for both source code and object code, so our method is of good adaptability.

The limitations of our method are listed below: (1) we can discover and locate the redundancy functionality and unreachable path, but can't determine whether the reduced redundancies are essential to or how many redundancies are suitable for system fault tolerance. Next, we'll try to solve this problem by large amounts of experiments to determine the empirical value of necessary redundancies; (2) as our method is static without the program run, the judgment of conditional predicate is difficult unless it's a constant expression. We plan to generalize our method to the dynamic analysis, considering the operating efficiency as well.

Compared with the ideal problem of code minimization decision, the study of this paper is preliminary. Consequently, there exists much more research work in this topic for us to launch in the future.

ACKNOWLEDGMENT

This research is supported by National NSF of China (No.91018008, No.61003268), Hubei Provincial NSF of China (No. 2010CDB08601) and the Fundamental Research Funds for the Central Universities (No.CUGL120283, No.3101038).

REFERENCES

[1] Nelson W. Accelerated Testing: Statistical Models, Test Plans and Data Analyses. Wiley Interscience, 1990.

[2] Formal Methods Specification and Verification Guidebook for Software and Computer Systems. Vol.1 Planning and Technology Insertion. Office of Safety and Mission Assurance, NASA, Washington DC, USA, NASA. GB.002.95, release 1.0 July 1990.

[3] Geffroy, J.C, G. Motet. Design of Dependable Computing Systems. Kluwer Academic Publishers. Dordrecht/Boston/London, 2002.

[4] www.pcworld.com/article/141226/open_source_security_bugs_uncovered.html

[5] Hennessy, J. L. and T. R. Gross. Postpass Code Optimization of Pipeline Constraints. ACM Transactions on Programming Languages and Systems 5(3), July 1983.

[6] Knoop, J, Ruthing,O.,and Steffen,B. Optimal code motion: Theory and practice. ACM Trans.Program.Lang.Syst.16, 4(July), 1117-1155. 1994.

[7] Keith D.Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In Proceedings of the 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), May 1999.

[8] Santosh Chede, Kishore Kulat "Algorithm to optimize code size and energy consumption in real time embedded system" in international journal of computers (Academy publisher), issue 3, July 2008.

[9] Zhang Dian, Research on Intermediate Code Optimization in GCC, Dissertation for the Master Degree in Engineering, Harbin University of Science and Technology, 2008.

[10] T. Simunic, L. Benini, G. D. Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In International Symposium on System Synthesis, 2000.

[11] E. Ozcan, and E. Onbasioglu, Memetic Algorithms for Parallel Code Optimization, International Journal of Parallel Processing, vol. 35, no.1/February (2007) 33–61.

[12] R.Kirner, P.Puschner, and A.Prantl, "Transforming flow information during code optimization for timing analysis," Journal of Real-Time Systems, vol.45, no.1-2, pp.72–105, Apr.2010.

[13] ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages (Austin, Texas, Jan. 24-26, 1983), ACM, New York, 177-189.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. ACM Trans. on Programming Languages and Systems, 9(3):319-349, July 1987.

[15] R.A.Ballance and A.B.Maccabe. Program dependence graphs for the rest of us. Technical Report CS92-10, Department of Computer Science, University of New Mexico, 1992.

[16] M.J. Harrold, B.A. Malloy, and G.Rothermel, "Efficient construction of program dependence graphs", Proceedings of the International Symposium on Software Testing and Analysis 93 (ISSTA93), pp. 160-70, June, 1993.

[17] PINTER, S. AND PINTER, R. Program optimization and parallelization using idioms. ACM Transactions on Programming Languages and Systems. Vol 16, NO 3, May 1994, Pages 305–327.

[18] Binkley, D., Harman, M., Krinke, J. Empirical study of optimization techniques for massive slicing. ACM Trans. Program. Lang. Syst. 30(1), 3(2007).

[19] C.Hammer and G.Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security, 2009.

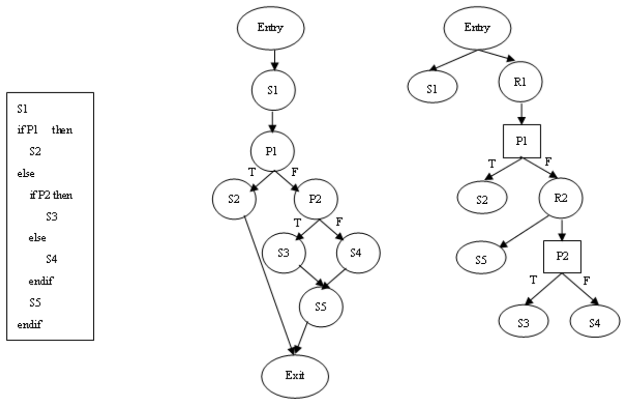
[20] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn and S. Yoo, "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem", Proceeding of the 3rd International Workshop on Search-Based Software Testing, 2010.

[21] R. E. Noonam, An Algorithm for Generating Abstract Syntax Tree, Computer Language, 3/4(10), 1985.

[22] Chris Eagle. IDA Pro Book. http://hex-rays.com/

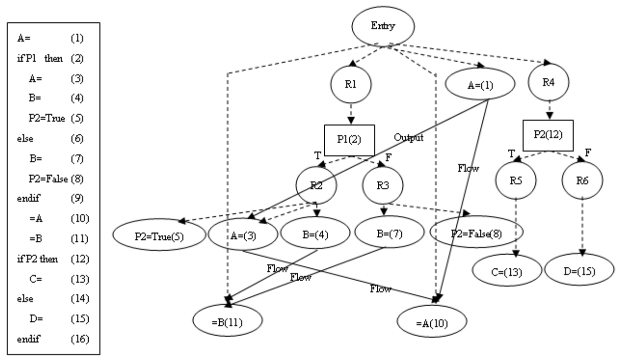
[23] LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flow graph. ACM Trans. Program. Lang. Syst. 1, 1 (July 1979), 121-141.

[24] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers", Computer Languages, 5, 1980, 141–153.



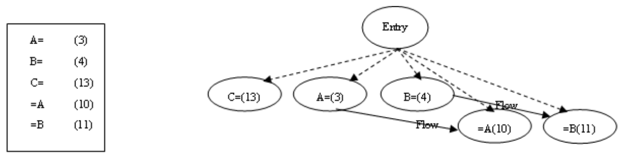
```
S1
if P1   then
   S2
else
   if P2 then
      S3
   else
      S4
   endif
   S5
endif
```

Figure 1. Control Flow Graph and Control Dependence Subgraph



```
A=          (1)
if P1 then  (2)
   A=       (3)
   B=       (4)
   P2=True  (5)
else        (6)
   B=       (7)
   P2=False (8)
endif       (9)
   =A       (10)
   =B       (11)
if P2 then  (12)
   C=       (13)
else        (14)
   D=       (15)
endif       (16)
```

Figure 2. Code Fragment and its PDG



```
A=   (3)
B=   (4)
C=   (13)
=A   (10)
=B   (11)
```

Figure 3. Simplified code fragment and its PDG after node merging and branch deletion