Aggressive Program Analysis Framework for Static Error Checking in Open64

Hongtao Yu Wei Huo ZhaoQing Zhang XiaoBing Feng
Key Laboratory of Computer System and Architecture, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing 100190, China
{ htyu, huowei, zqzhang, fxb }@ict.ac.cn

ABSTRACT

Nowadays error checking becomes more and more significant for constructing high reliable software. In this paper, we will introduce our work of integrating static error checking into Open64. We are devoting to construct an aggressive program analysis framework for error checking in the compiler. We integrate the intraprocedural analysis into interprocedural phase in order to do flow- and context-sensitive whole program analysis. The precision of alias information can heavily impact many consequent analyses and error checking. We also have improved the original alias analysis to be field-sensitive so that field members of the same structural object can be distinguished in the resulted points-to graph

Keywords

Program analysis, pointer analysis, error checking

1. Introduction

Open64 is a high performance compiler for generating effective binary codes. Up to now it can generate code for the Intel IA-64, Intel IA-32e and AMD X8664 architecture. It derives from the SGI compiler for the MIPS R10000 processor, called MIPSPro. It was released under the GNU GPL in 2000, and now mostly serves as a research platform for compiler and computer architecture research groups. Open64 supports Fortran 77/95 and C/C++, as well as the shared memory programming model OpenMP. It can conduct high-quality interprocedural analysis, data-flow analysis, data dependence analysis, and array region analysis.

However, existed scalar analysis of Open64 is not precise enough to serve for static error checking since it is originally designed for optimizations. The original interprocedural framework is to summary each procedure flow-insensitively first, and then transfers the summarized information on the procedure call graph iteratively. This framework limits us to perform flow-sensitive interprocedural analysis inconveniently. For example, we cannot obtain the must modified side effect information of a single procedure since we cannot determine whether a global variable is modified on all paths from entry to exit in the local control flow in the summary phase. Also the original framework is field-insensitive. Although the intraprocedural data-flow analysis is field-sensitive, the intraprocedural alias analysis is field-insensitive. Field-insensitive alias analysis avoids us obtaining field-sensitive mod/ref information, which means that we cannot make sure whether a specific field of a structural object is modified or referenced by a procedure.

Our aim is to improve the interprocedural phase in order to obtain more precision in analysis, and to be more accurate in error checking furthermore. To gain flow-sensitivity, we integrate the intraprocedural analysis phase into interprocedural phase, allowing the interprocedural analysis to access and update local control-flow and data-flow information flexibly. To achieve context-sensitivity, transfer functions modeling procedure effects are computed for each procedure, and are used in each call site. Several data-flow problems, e.g. pointer analysis, const propagation, program slicing and etc, can be solved in a common framework called modular interprocedural analysis such as in [2] that is composed of two passes of analyzing the whole program. The first pass traverses on the procedure call graph in a bottom-up order, computing transfer functions for each procedure. The second pass traverses on the call graph in a top-down order, propagating data flow values from the entry to the exit of each local control flow graph. Transfer functions are used to get the result of procedure side effect when the top-down phase propagates data flow values over a function call. Field-sensitivity is obtained by treating each field of any structural object as an individual variable in all kinds of analysis. Fields in the same structure are distinguished in the form of offset from its base structure and its data type size according to the target machine information.

Under the new framework, statically checking common programming errors are developed. We abstract several kinds of error checking problems as a common dataflow problem that can be solved using the fix point theory on a semi-lattice [3]. Consequently the error checking problem is treated as a common data flow analysis under the framework. We have developed a program template that concentrates on flow- and context-sensitive interprocedural dataflow analysis by using C++ generic programming.

We have also improved the original pointer analysis of Open64 to be field-sensitive. We have provided two kinds of pointer analysis. One is directly implemented from [1], and the other is designed by ourselves in which we consider the target machine ABI in the intermediate representations.

The former one achieves more precision and higher efficiency than the original one. The latter one achieves more precision than the former one while maintains almost the same efficiency as it.

Our main contributions to Open64 are that we have designed and implemented:

- A flow- and context-sensitive interprocedural framework under which kinds of error checking are performed.
- Two kinds of efficient field-sensitive pointer analysis.

The rest of this paper is organized as follows: In Section 2, we introduce the new framework and the work we have done in Open64 briefly. In Section 3 we describe the pointer analysis. In Section 4, we propose the flow- and context-sensitive interprocedural analysis method in detail and its application in error checking. We survey related work in Section 5 and conclude in Section 6.

2. Framework

The new framework is displayed in Figure 1. It starts after the phase IPA_LINK linking all WHIRL representations together. The interprocedural analysis produces useful information for the consequent client error checking.

The analysis starts with a field-sensitive unification-based pointer analysis [1] which is flow- and context-insensitive to locate the target of function pointers. Once the targets of function pointers are determined, we begin to build a procedure call graph in which an indirect call site through a function pointer is modeled as a branch of direct calls to its target. We intend to design and implement a pointer analysis architecture that employs several kinds of field-sensitive algorithms that differ in precision and efficiency. The series of pointer analysis are performed in the increasing order of precision. Each analysis is performed on the base of the former analysis so that we can obtain higher efficiency than performing the analysis separately. The first pointer analysis of the pointer architecture is the field-sensitive unification-based pointer analysis. Other field-sensitive flow- and context-insensitive pointer analysis may also be performed after this, e.g. inclusion-based pointer analysis [5] [6] to make the result points-to graph more precise and further refine the call graph. Up to now, only the field-sensitive unification-based pointer analysis has been implemented.

The interprocedural control flow optimization includes Dead Function Elimination (DFE) and Fake Control Flow Elimination (FCFE). FCFE recognizes the program points where control flow must terminate, i.e. exit the whole program, and eliminates the fake control flow starting from these points. For example, the “abort” function and “exit” function in C/C++ language can cause the running process to terminate. Recognizing these program points can help us avoid control flow reaching the consequent program points immediately after the terminated point, and furthermore eliminate fake data flow. We define *termination procedure* as a procedure which may terminate the running process in sequential programs. The termination procedure in C/C++ does not contain only the “abort” and “exit” function, but also consists of any wrapper procedure that invokes termination procedures on one path of its control flow graph. Thereby, recognizing termination procedures is an interprocedural flow- and context-sensitive problem. We compute transfer function for each procedure that under which input the procedure must terminate. We create individual control flow graph for each procedure and refine it after FCFE.

After the interprocedural control flow optimization, we start to construct the Static Single Assignment (SSA) form [7] for each procedure. The data flow analysis infrastructure in WOPT has been migrated to IPA phase, allowing us to do data flow analysis flexibly. The result of pointer analysis is used to make SSA form more precise. Also for the sake of precision, zero versioning and virtual variable techniques [8] for building SSA are improved. Another way to construct SSA form is to perform the flow- and context-sensitive pointer analysis which analyzes the points-to sets of variables in each program point while building SSA form. The flow- and context-sensitive pointer analysis is now under developing and will be introduced in another article.

3. Field-sensitive pointer analysis

The field-sensitive pointer analysis is unification-based and is improved from [1]. We have implemented the original algorithm in [1] as well as our improved one and obtain the performance results in Table 1 and Table 2.

Table 1 displays the comparison between the current alias analysis of Open64 which directly implements the algorithm in [4] and the algorithm in [1]. The former one is field-insensitive. The example benchmarks are all taken from SPEC2000.

The columns of the table identify:

- *Example* : the benchmark name ;
- *KLOC*: the size of the benchmark (line numbers counted by kilo lines);
- *Field OPs*: the number of indirect memory access to fields of structural objects;

  Classes: the number of alias classes of the total *Field OPs* above; memory access operations in the same alias class are regarded as aliased, viz. they access the same memory location.
- *Max*: the maximal number of *Field OPs* in the same alias class;
- *Min*: the minimal number of *Field OPs* in the same alias class;
- *Average*: the average number of *Field OPs* in the same alias class, Calculated by division *Field OPs* by *Classes.*
- *Time*: the time for analyzing the benchmark;

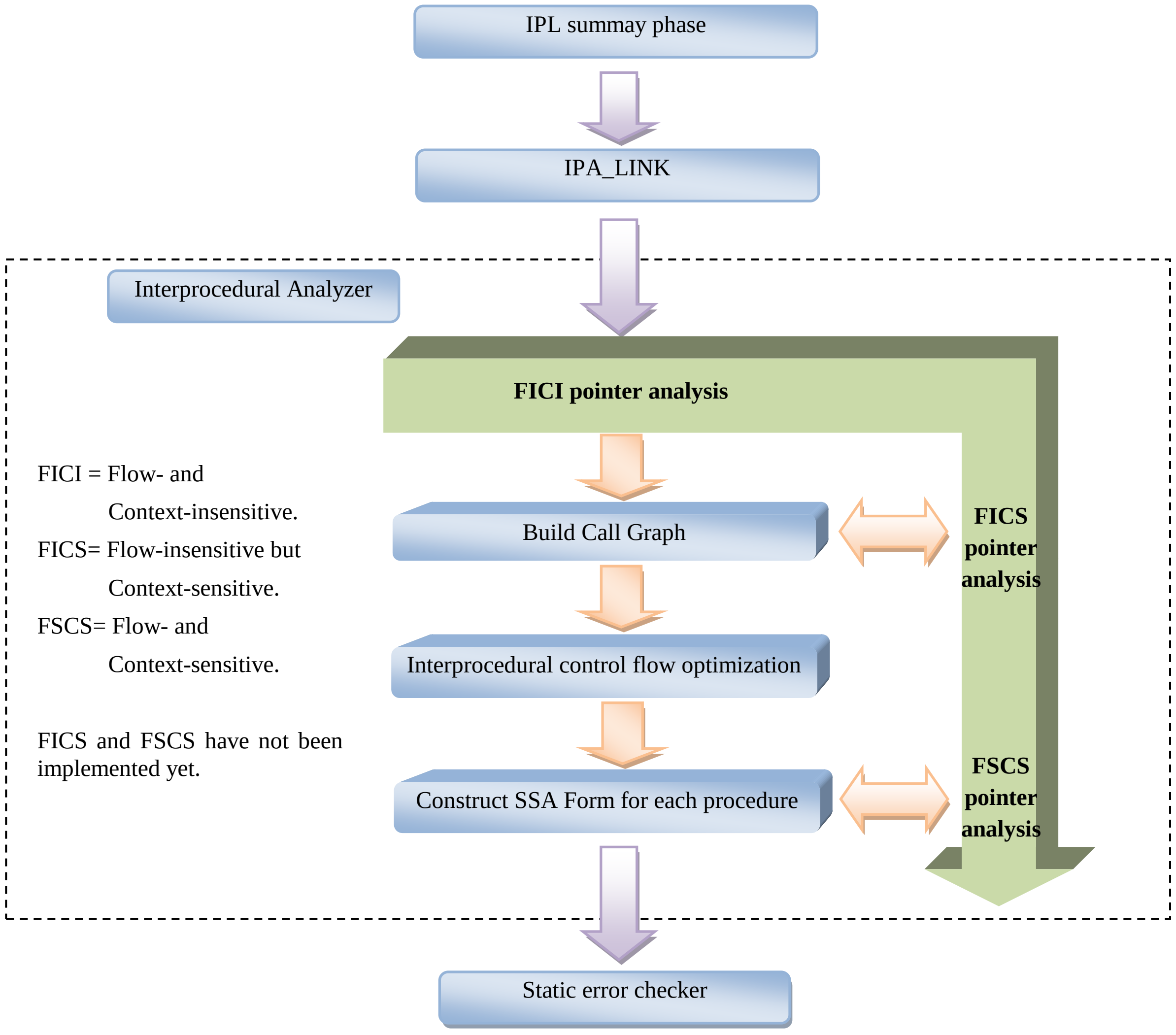

IPL summay phase
IPA_LINK
Interprocedural Analyzer
FICI pointer analysis
FICI = Flow- and Context-insensitive.
FICS= Flow-insensitive but Context-sensitive.
FSCS= Flow- and Context-sensitive.
FICS and FSCS have not been implemented yet.
Build Call Graph
FICS pointer analysis
Interprocedural control flow optimization
Construct SSA Form for each procedure
FSCS pointer analysis
Static error checker

Figure 1. The flow- and context-sensitive framework in Open64.

To our surprise, the analyzing time of field-insensitive Steensgaard classification is more than the field-sensitive one. We trace the analyzing process and find out that the exceeding time is spent on the union-find operation since object pointed to by all undistinguished fields must be joined. The latter is more precise and efficient than the former one.

Table 2 shows the comparison between the algorithm in [4] and our improved algorithm. They are both field-sensitive. The improved algorithm is precise than the original one while is a little less in efficiency. The main idea of improvement is to consider memory layout in high-level analysis in order to precisely distinguish fields of structure objects. In our improved method, a field in the program is represented by a pair of offset from its base structure and size of its own data type. The implementation of this field representation makes use of the target ABI information on WHIRL representation. Furthermore, we have lowered all the structural memory operations to a series of scalar memory operations based on the target machine information, allowing consequential data flow analysis more aggressive. The method is also portable due to the architecture of Open64.

We take a small program as an example to indicate intuitively that considering memory layout in the high-level analysis is more aggressive. The example showed in Figure 3(a) is taken from [1]. Figure 3(b) displays the result points-to graph of analysis in [1]. Figure 3(c) shows the result points-to graph of our improved algorithm for the target machine X8664.

Example	KLOC	Field Ops	Field-insensitive Steensgaard Classification					Field-sensitive Steensgaard Classification				
			Classes	Max	Min	Average	Time (secs)	Classes	Max	Min	Average	Time (secs)
mcf	0.9	562	7	527	1	80.28	0.02	34	376	1	16.52	0.02
bzip2	2.4	87	2	69	18	43.5	0.03	2	69	18	43.5	0.03
gzip	3.6	244	4	222	4	61	0.05	37	72	1	6.59	0.05
parser	7.4	2375	18	2115	3	131.94	0.13	61	2003	1	38.93	0.11
vpr	8.7	1649	23	1408	2	71.69	0.17	159	955	1	10.37	0.14
crafty	9.7	830	6	268	30	138.33	0.5	74	268	1	11.21	0.22
twolf	15	6457	1	6457	6457	6457	0.48	80	4495	1	80.71	0.23
vortex	25	8062	90	7631	1	89.57	1.17	274	7616	1	29.42	0.60
gap	28	11480	9	11459	1	1275.55	1.04	134	11023	1	85.67	0.60

Table 1. Comparing current alias analysis in Open64 with the analysis in [1]

Example	KLOC	Field Ops	Field-sensitive Steensgaard Classification					Aggressively Field-sensitive Classification				
			Classes	Max	Min	Average	Time (secs)	Classes	Max	Min	Average	Time (secs)
mcf	0.9	562	34	376	1	16.52	0.02	69	39	1	8.14	0.02
bzip2	2.4	87	2	69	18	43.5	0.03	7	26	6	12.42	0.03
gzip	3.6	244	37	72	1	6.59	0.05	44	49	1	5.54	0.05
parser	7.4	2375	61	2003	1	38.93	0.11	88	1989	1	26.98	0.13

Table 2. Comparing the analysis in [1] with our improved method

The increase of precision also benefits from improving the type hierarchy proposed in [1] which defined the partial order among scalars and structured objects. The original type hierarchy is showed in Figure 2(a). The type hierarchy provides a necessary requirement for partial order $a \trianglelefteq_s b$ to hold is that a and b are either of the same kind or the kind of b appears above the kind of a in the hierarchy [1]. It results in the conditional $\sqsubseteq_s$ join and conditional join $\trianglelefteq_s$ of *simple* type and *struct* type conservatively, since both of them must be promoted to *object* type. We make a change for the type hierarchy and corresponding change in type system that enables the result of joining *simple* type and *struct* type is another *struct* type with only fields possibly overlapped with the scalar joined. The new hierarchy is showed in Figure 2(b).

4. Flow- and context-sensitive dataflow analysis

In this section, we will introduce our flow- and context-sensitive engine for global data flow analysis and its one application. We have implemented the data-flow engine in C++ template for the sake of code reusing. The engine consists of two components, one is a transfer function evaluator and the other is a dataflow value propagator.

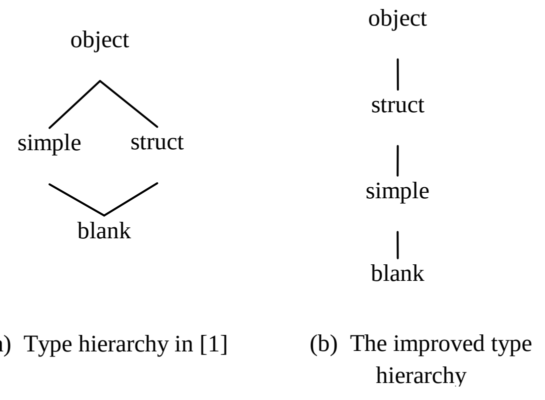
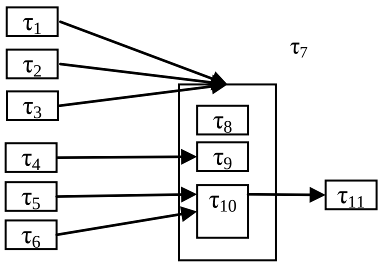


Figure 2. The comparison in type hierarchy

The transfer function evaluator computes transfer functions for each procedure which summarize the effect of this procedure under a specific data flow problem. A single transfer

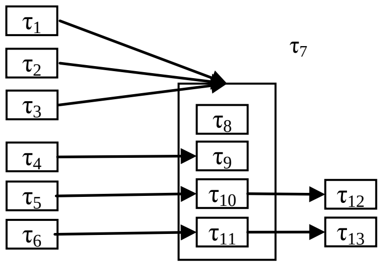
```
int    i1, *i2, **i3, **i4;
float  f1, **f2;
struct { int  a, *b, *c; }  s1, *s2;
struct { int  d, *e; float  f, *g }  s3, *s4;
s2 = &s1;
s4 = &s3;
f2 = &s4->g;
*f2 = &f1;
i3 = &s2->b;
i4 = &s2->c;
*i4 = &i1;
i2 = (int*) s2;
i2 = (int*) s4;
```

(a) a small C program



i2:	τ_1	s1.a:	τ_8
s2:	τ_2	s3.d:	τ_8
s4:	τ_3	s1.b:	τ_9
i3:	τ_4	s3.e:	τ_9
i4:	τ_5	s1.c:	τ_{10}
f2:	τ_6	s3.f:	τ_{10}
s1:	τ_7	s3.g:	τ_{10}
s3:	τ_7	i1:	τ_{11}

(b) points-to graph of analysis in [1]



i2:	τ_1	s1.a:	τ_8
s2:	τ_2	s3.d:	τ_8
s4:	τ_3	s1.b:	τ_9
i3:	τ_4	s3.e:	τ_9
i4:	τ_5	s1.c:	τ_{10}
f2:	τ_6	s3.f:	τ_{11}
s1:	τ_7	s3.g:	τ_{11}
s3:	τ_7	i1:	τ_{12}

(c) points-to graph of improved alogorithm for the IA-32 target

Figure 3. Intuitively compare work in [1] with our improved work.

function has the form $y = f(x_1, x_2 ... x_n)$, in which y denotes a formal-out parameter and $x_1 \ldots x_n$ denote a list of formal-in parameters. A formal-in parameter of a procedure is either a declared formal parameter or a location whose value at the procedure entry may be accessed by the procedure or the procedures it invokes. Such location may be a global variable, an allocated object or represented by dereference of pointers. Similarly, the formal-out parameters of a procedure include not only the return value of this procedure but also all the locations whose value at the procedure exit may be accessed out of the procedure. The function body of f gives the mapping relations between inputs $x_1 \ldots x_n$ and the output y. Since a procedure may have more than one formal-out parameters, it may have a group of transfer functions each of which summarizes the procedure effect for one of the formal-out parameters.

We compute transfer functions for each procedure over the procedure call graph in a bottom-up order, from callees to callers. When computing a caller's transfer function, all callees' transfer functions have been already computed and are compounded into the caller's transfer function if there is no circle in the call graph. To handle recursions, we reduced the call graph to a SCC-DAG (Strongly Connected Component Directed Acyclic Graph) and perform iterations in each SCC.

The dataflow value propagator propagates dataflow value from the entry to the exit of each procedure's local control flow graph, and propagates on the call graph in a top-down manner. A data flow value is an element of the semi-lattice corresponding to different data flow problems. At the entry of each procedure, we perform the "meet" operation on the data flow values from difference call sites for each formal-in parameter. At each callsite, we propagate value of each actual-in parameter to the corresponding formal-in parameter of the callee at first. Then we obtain the value of each formal-out parameter by applying the transfer function to the value of formal-in parameters and propagate it to the corresponding actual-out parameter. If there are recursions in the program, we also need to perform iterations in each SCC. We have provided a generic algorithm for solving forward dataflow problems. The pseudo code is displayed in Figure 4. The intraprocedural algorithm traverses local control flow graph in topological order while manipulating a dataflow value stack for each variable. The stack records a sequence of dataflow value that stands for the dataflow trace by merging all paths from entry to the current program point. The top value of the stack is the current value in the program point. When processing an assignment statement, the result dataflow value is pushed into the stack.

Our first application of the engine is to find all the possible references of uninitialized memory. Variables or allocated memory objects can assume unexpected values if they are used before they are initialized. Referencing uninitialized memory may cause a program to behave in an unpredictable or unplanned manner.

4.1 Checking uninitialized reference

We abstract the task as solving a dataflow problem in which any memory object in any reference site has as a property the dataflow value "define", "may define" or "undefine". If a memory object is initialized on all paths from the program entry (usually as "main" function) to the current reference site, the memory object in this site has the property "define". If on some path the memory object is not initialized, the property will be "undefine". The initialization through indirect memory operations (e.g. the deference of pointers) results in a property "may define".

To solve the problem we first invoke the transfer function

```
/* Interprocedura dataflow value propagator */
Dataflow_value_Propagator(call_graph )
Begin
  for each node scc of call_graph's SCC_DAG in
     topological order
    if scc is not a recursive cycle and contains procedure proc
      call Propagate_on_proc( proc)
    else
      changed ←true;
      while ( changed )
        changed ← false;
        for each proc of scc do
          changed ∨←
              call Propagate_on_proc ( proc );
End
Propagate_on_proc ( proc )
Begin
  changed ←false;
  for each node scc of CFG SCC_DAG in
     topological order
    if scc is not a recursive cycle and contains basic block bb
      changed ∨←
              call Propagate_on_ bb ( bb );
    else
      changed ←true;
      while ( changed )
        changed ← false;
        for each bb of scc do
          changed ∨←
              call Propagate_on_loop ( bb );
  return changed;
End
```

Figure 4 The algorithm for interprocedural data flow value propagation

evaluator to compute a transfer function for each procedure. The transfer function of one procedure says that which global variables are modified by this procedure. In fact the transfer function evaluator performs an interprocedural modified side effect analysis to the whole program. The side effect analysis can distinguish "must" and "may" mod information:

i. If on every path in the procedure form entry to exit the global variable is modified by direct assignments, the variable is "must mod".
ii. Otherwise if in every path variable is modified by either direct or indirect assignments, the variable is "may mod".

iii. Otherwise the variable must not be modified on some of the paths, so it is "may not mod"

We compute the transfer function of a procedure by building SSA form for the procedure first. If there are multiple exit nodes in the CFG, we connect all the exits to a unique exit node, called "fake exit". We insert a SSA Φ-function in the "fake exit", called "exit Φ", for each global variable whose value is modified or referenced in the procedure body or in the body of procedures invoked by this procedure. After building SSA form, we start to find the possible definitions of each global variable appeared in the exit Φ backwards from the fake exit to the entry in order to determine the modified property for the variable.

The dataflow value propagator propagates the modification property at procedure entry to the exit. The "must mod" and "may mod" value of a transfer function are both regarded as "define" in this phase. The "may not mod" value is regarded as "undefine" here. We only propagate for scalar variables that are either local or global currently. The experimental results are displayed in Table 3. The columns of the table identify:

- *Example* : the benchmark name ;
- *KLOC*: the size of the benchmark (line numbers counted by kilo lines);
- *Time*: the time for checking the benchmark;
- *Reports*: the total number of warnings produced on the benchmark;
- *Bugs*: the number of true bugs found;
- *FP Rate*: the false positive rate;

*Exam*ple	Kloc	Time (sec)	Reports	Bugs	FP Rate
mcf	0.9	0.01	3	3	0%
bzip2	2.4	0.05	0	0	0%
bftpd-2.3	2.8	0.08	2	1	50%
gzip	3.6	0.11	1	1	0%
HyperSAT-1.7	6	0.11	1	0	100%
parser	7.4	0.7	0	0	0%
TOTAL	23.1	1.06	7	5	28.6%

Table 3. The uninitialized reference checking of scalar variables

We have used compiler GCC with the warning option –Wuninitialized to check the uninitialized reference for the examples in Table 3. However no warnings are reported. This is mainly because GCC can only check uninitialized reference for auto variables intraprocedurally.

5. Related work

In the literature of pointer analysis, much work has been devoted into improving the precise and efficiency from flow-sensitivity and context-sensitivity [2, 4, 5, 9-13]. However, field-sensitivity is somehow not taken seriously as the former two properties since there is not so much work in the literature. Due to the widely uses of aggregate objects in practical programs, the field-sensitivity can greatly impact the precision of pointer analysis. Steensgaard provided a field-sensitive version [1] of his original work [4]. Yong et al. proposes a framework of field-sensitive points-to analysis that covers various levels of field-sensitivity [14]. They use separated variables to represent different fields. Their framework is designed to be able to handle type casing presenting in many C programs. Pearce et al. present a novel approach for precisely modeling structural variables and indirect function calls [6]. In their work field reference expressions are translated into a form of a variable plus a field id in the constraint system. Whaley's work [12] is also field-sensitive. However, his method was originally designed for Java and he did not discuss how to handle type casting.

Mygcc [15] is the first checking technique to combine checking and compiling together in order to do permanent checking. They call the checking is permanent because programmers can check programs while compiling the programs, avoiding extra time cost. Integrating checking to compiling also enables a software development method in which checking permanently accompanies evolution because compiling is always necessary in coding phase. However, their approach is not interprocedural. Other checking tools like Fast check [16], Saturn [17], Calysto [18] is not permanent.

6. Conclusion and future work

We have introduced our work of constructing an aggressive framework for program analysis in order to do error checking in Open64. The framework integrates intraprocedural analysis into interprocedural phase so that we can do flow- and context-sensitive whole program analysis. We have also improved the original alias analysis to be field-sensitive and compared the three unification-based methods.

In the future, we are going to finish the pointer analysis architecture. We intend to implement a field-sensitive inclusion-based points-to analysis immediately after the unification-based method. To make the analysis more scalable, we will adopt the bootstrapping strategy, inspired by [13] that partitions the whole program into many slices. Statements that directly modified pointers in the same alias class will be put into the same slice. If we want to obtain the inclusion-based result of a pointer, it suffices to restrict the inclusion-based analysis only to the slice that directly modified the pointer and the slices that modify pointers which may point to the pointer. Since Anderson-styled pointer analysis has the cubic time complexity, the bootstrapping method makes use of a simple formula to improve its efficiency:

$$(x_1 + x_2)^3 \geq x_1^3 + x_2^3.$$

The larger the program is, the more analysis time we can save. We have not implemented it yet.

Benefiting from the unification-based points-to analysis, we have designed a scalable flow- and context-sensitive pointer analysis. The analysis takes advantage of the points-to graph given by the unification-based analysis to incrementally build SSA form while analyzing points-to set of each variable. We called the analysis “hierarchical analysis” since we analyze pointers in the increasing order of Steensgaard pointer graph. The analysis is under development currently.

7. Reference

[1] Steensgaard, B. Points-to Analysis by Type Inference of Programs with Structures and Unions. In *Proceedings of the Proceedings of the 6th International Conference on Compiler Construction*. Springer-Verlag, 1996.

[2] Cheng, B.-C. and Hwu, W.-M. W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM, Vancouver, British Columbia, Canada, 2000.

[3] Muchnick, S. S. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[4] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, St. Petersburg Beach, Florida, United States, 1996.

[5] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language* PhD., University of Copenhagen, DIKU, 1994.

[6] Pearce, D. J., Kelly, P. H. J. and Hankin, C. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30, 1 2007), 4.

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13, 4 1991), 451-490.

[8] Chow, F. C., Chan, S., Liu, S.-M., Lo, R. and Streich, M. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proceedings of the Proceedings of the 6th International Conference on Compiler Construction*. Springer-Verlag, 1996.

[9] Chase, D. R., Wegman, M. and Zadeck, F. K. Analysis of pointers and structures. In *Proceedings of the Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM, White Plains, New York, United States, 1990.

[10] Wilson, R. P. and Lam, M. S. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. ACM, La Jolla, California, United States, 1995.

[11] Chatterjee, R., Ryder, B. G. and Landi, W. A. Relevant context inference. In *Proceedings of the Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, San Antonio, Texas, United States, 1999.

[12] Whaley, J. and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM, Washington DC, USA, 2004.

[13] Kahlon, V. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, Tucson, AZ, USA, 2008.

[14] Yong, S. H., Horwitz, S. and Reps, T. Pointer analysis for programs with structures and casting. In *Proceedings of the Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM, Atlanta, Georgia, United States, 1999.

[15] Volanschi, N. A Portable Compiler-Integrated Approach to Permanent Checking. In *Proceedings of the Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2006.

[16] Cherem, S., Princehouse, L. and Rugina, R. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM, San Diego, California, USA, 2007.

[17] Dillig, I., Dillig, T. and Aiken, A. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, Tucson, AZ, USA, 2008.

[18] Babic, D. and Hu, A. J. Calysto: scalable and precise extended static checking. In *Proceedings of the Proceedings of the 30th international conference on Software engineering*. ACM, Leipzig, Germany, 2008.