Towards A Structural Load Testing Tool

Cheer-Sun D. Yang and Lori L. Pollock

Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
(302) 831-1953

Abstract

Load sensitive faults are programming errors with which the program executes successfully if executed for a short time or small workload, but cause the program to fail when it is executed under a heavy load or continuously. Example applications that are prone to load sensitive faults include networking programs, client-server programs, operating system modules, and embedded systems. An important component of testing these programs is *load* or *stress testing,* which is directed towards detecting load sensitive faults, and also often performed to insure that a system satisfies a particular performance requirement under a heavy load.

Current approaches to load testing treat the system as a *black box*, generating test data based on parameters specified by the tester within an operational profile. This method of load testing is applied at system integration time, typically resulting in a high maintenance cost in response to detecting a fault. This method can be applied only when the distribution of input data follows some probability distribution. In addition, because it is a functional testing approach, it may not execute every load sensitive path extensively.

In this paper, we advocate a structural approach to load testing, which analyzes the program code for each module to produce test data for load testing. There exist many structural testing methods; however, their main goal is generating test data for executing all statements, definition-use pairs, or paths of a program at least once, without consideration for executing any particular path extensively. Thus, previous work on structural testing does not subsume the problem of structural load testing.

Our work has focused on the identification of *load sensitive* modules based on a conservative static analysis of the module's code. A load sensitive module is one with the property that its correctness depends on the amount of input data or the length of time that the program will execute continuously. We have developed an algorithm for computing a *load sensitivity index* for various points throughout the program module, using a control flow graph representation of the program. Based on this load sensitive index, we identify a program module as either *must-be load sensitive*, *may-be load sensitive*, or *load insensitive*.

By developing a structural approach to load testing, we can perform load testing at the stage of program development rather than system integration. Thus, detected load sensitive faults can be fixed more cheaply. In addition, by identifying the load sensitive modules, we can limit the test data to focus on these modules, and reduce the amount of load testing that is performed. Our ultimate goal is to develop a testing tool for structural load testing which takes a program as input, and uses our techniques to automatically determines whether that program needs to be load tested, and if so, automatically generates test data for structural load testing of the load sensitive program.

1 Introduction

Load sensitive faults are programming errors with which the program executes successfully if executed for a short time or small workload, but cause the program to fail when it is executed under a heavy load or continuously. Two common situations under which load sensitive faults may occur are memory leaks and incorrect dynamic memory allocation. Example applications that are prone to load sensitive faults include networking programs, client-server programs, operating system modules, and embedded systems. An important component of testing these programs is *load* or *stress testing,* which is directed towards detecting load sensitive faults, and also often performed to insure that a system satisfies a particular performance requirement under a heavy load.

Figure 4 depicts a segment of code reflecting a situation in which a dynamic character string may need space longer than what was allocated. When this condition occurs, another dynamic allocation is invoked, the original string is copied into the new location, and the original space is supposed to be deallocated. However, in this case, the programmer has forgotten to include the deallocation. As a result, this code segment may cause the program to fail when it is executed continuously, and the true condition is satisfied many times.

Current approaches to load testing treat the system as a *black box*, generating test data based on parameters specified by the tester within an operational profile. One particular method is called Deterministic Markov State Testing [1],[2]. In this approach, the input data is assumed to arrive according to a Poisson distribution, is serviced in accordance to an exponential distribution, and the system being tested is considered as a black box. These assumptions limit the application of this approach to applications that can be modeled by using the Markov Chain. In addition, because it is a functional testing approach, it may not execute every load sensitive path extensively. The functional approach may fail to detect the fault in Figure 4 if the size *s1* is large enough, such that the true condition of the if-statement will not be satisfied during the functional testing. As a result, the program will always operate successfully. Moreover, this method of load testing is applied at system integration time, typically resulting in a high maintenance cost in response to detecting a fault.

In this paper, we advocate a structural approach to load testing, which analyzes the program code for each module to produce test data for load testing. That is, this approach is code-based, whereas the functional load test is specification-based. There exist many structural testing methods; however, their main goal is generating test data for executing all statements, definition-use pairs, or paths of a program at least once, without consideration for executing any particular path extensively [10],[5], [3],[9], [7],[8]. Thus, previous work on structural testing does not subsume the problem of structural load testing. In addition, although a careful code inspection by human may find faults like the simple segment in Figure 4 at the stage of code development, this situation would undoubtedly be missed when coded inside a complicated program.

Our work has focused on the identification of *load sensitive* modules based on a conservative static analysis of the module's code. A load sensitive module is one with the property that its correctness depends on the amount of input data or the length of time that the program will execute continuously. We have developed an algorithm for computing a *load sensitivity index* for various points throughout the program module, using a control flow graph representation of the program. Based on this load sensitive index, we identify a program module as either *must-be load sensitive*, *may-be load sensitive*, or *load insensitive*.

```
typedef struct {
   int a;
   int b;
   char c[100];
} DATA;

DATA *pt;

main()
{
    while(1)
    {
        pt = (DATA *)malloc(sizeof(DATA));
        pt->a = 10;
     }
}
```

Figure 1: A C Code Example

By developing a structural approach to load testing, we can perform load testing at the stage of program development rather than system integration. Thus, detected load sensitive faults can be fixed more cheaply. In addition, by identifying the load sensitive modules, we can limit the test data to focus on these modules, and reduce the amount of load testing that is performed. Our ultimate goal is to develop a testing tool for structural load testing which takes a program as input and uses our techniques to automatically determine whether that program needs to be load tested, and if so, automatically generates test data for structural load testing of the load sensitive program.

The rest of this paper is organized as follows. In Section 2, our structural load testing approach is outlined. Section 3 defines the load sensitivity index, and briefly presents our algorithm for computing the LSI for a module and among modules. Finally, Section 4 summarizes our initial work and discusses our current and future work.

2 The Structural Load Testing Approach

Our overall process for structural load testing consists of three steps. This paper focuses on the first of these 3 steps of the structural load testing approach.

(1) Identify the degree of load sensitivity of the target program. This step consists of computing the LSI for the target program, and based on that index, determining (a) whether the program needs test cases for

```
class String
{
  char *str;

public:
  String() { str = new char; *str = 0; }
  String(char *);
  void print();
  ~String() { delete str; }
};


String::String(char *s)
{
  str = new char[strlen(s) + 1];
  strcpy(str, s);
}

void
String::print()
{
  str[strlen(str)] = '\0';
}

main()
{
  while(1)
  {
     String *sp = new String("hello world");

     sp->print();
  }
}
```

Figure 2: A C++ Example

```
DATA *pt;
int i;
int worker();

real_main()
{
   eval("worker", worker());
    i = 0;

    while(1)
    {
       pt = (DATA *)malloc(sizeof(DATA));
       pt->a = 10;
       if( i < LIMIT )
          out("data", i, pt->a);
       else
        {
           out("data", i, 1);
           break;
        }
        i++;
    }
}



worker()
{
  int 1;
  int a;

  while(1)
  {
     rd("data", i, ?a);
     if ( i == LIMIT)
        break;
     else
        printf("The data received is %d\n", a);
     i++;
  }
}
```

Figure 3: A C-Linda Example

```
char *pointer1, *pointer2;
while( TRUE )
{
  pointer1 = malloc(s1);        /* s1 is the size allocated */
   ...

  if (s1 is too small)
  {
     pointer2 = malloc(s2);        /* s2 > s1 */

       ...
     strcpy(pointer2, pointer1);
     pointer1 = pointer2;
  }
   ...
  free(pointer1);
}
```

Figure 4: A Buggy Load Sensitive While-loop

load testing, and if so, (b) using the LSI of parts of the program to drive the test case generation.

(2) Automatically generate test data. Our goal is to generate test cases for testing the load sensitive loops extensively, taking into account boundary conditions and average conditions. Therefore, the worst case, best case, and the average case LSI's will be analyzed. Based on these indices, test data will be automatically generated.

(3) Develop a computer-aided tool. The ultimate end product of this research is to develop a tool for the purpose of structural load testing. This tool would be used to visualize the process of testing.

3 Computing the Load Sensitivity Index

Our initial work on structural load sensitivity has focused on the detection of load sensitive faults caused by incorrect implementation of dynamic allocation of memory. In particular, when dynamic allocation is used within an infinite loop, the heap space will eventually overflow if proper deallocation is missing. To determine the load sensitivity of a program, we define a *Load Sensitivity Index (LSI)* to indicate the net increase or decrease of heap space used for each iteration of an infinite loop, that is a loop in the program which purposely runs continuously. More specifically, the LSI is defined to be the size of the heap space allocated minus the size deallocated during one iteration of the infinite while- or for-loop.

Weyuker (1979) has shown that it is not decidable whether or not a path, a statement, or a given branch will be executed. Thus, we cannot always obtain precise information through a static analysis of the program for computing the LSI. However, we need to insure that our analysis is conservative so we do not miss load testing a program that should be load tested. In particular, we compute the worst case and the best case LSI's to be the range of possible LSI values. Due to the conservativeness of the analysis, a program may not be load sensitive at run-time, when our algorithm claims that it is classified as *MAY-BE load sensitive.*

In order to automate the process of computing LSI, some assumptions are required. Since the program being tested will be executed continuously for a long time, we assume that an infinite while-loop or for-loop always exists in the CFG, and is easily identified. We assume that *programmers write programs that are nearly correct.* In other words, a programmer will not intentionally plant coding errors in a program.

Loops of unknown bounds are assumed to execute a constant number of times, where the constant that we use for the analysis is conservatively chosen for worst and best case analysis. In particular, for worst case analysis, loops of unknown bounds for which the LSI for a single iteration is positive are assumed to be executed a large number of times, while loops for which the LSI for a single iteration is negative are assumed not to be executed. For best case analysis, the opposite holds. Without any further data flow analysis to determine when two loops have the same unknown bound, one loop being positive and the other negative, this is an overly conservative assumption. We are currently investigating less conservative assumptions based on previous analysis of the program's loops.

We first describe the program representation upon which the computation is performed, followed by our algorithm for computing the LSI of a program, and finally how we use this index to classify a program's degree of load sensitivity.

3.1 Annotated Control Flow Graph Representation

To perform a structural analysis of a program, the natural representation of the program is a graphical form that explicitly depicts the possible paths of execution. The most common graphical representation of a program, and the one produced at some time during compilation by almost every compiler, is the *control flow graph* (CFG). A CFG is a graph G=(V, E), where **V** is the set of nodes representing basic blocks of a program, and **E** is the set of edges representing the possible flow of control between these straight-line segments of code.

In order to determine the load sensitivity of a program, we need to annotate CFG nodes with some information for computing load sensitivity. For each *malloc()* call, we need to store the size of the heap space to be allocated. This size information can be obtained by examining the expression passed as the parameter to the *malloc()*. Similarly, for each *free()* call, we need to store the size of storage to be deallocated. The size information is obtained by examining the definition of the variable being freed. Gathering precise information about these sizes of allocations and deallocations may require data flow analysis to determine the use-def and def-use information and type information. This is straightforward data flow analysis, typically performed by an optimizing compiler.

3.2 Algorithm for Computing the LSI

In order to compute useful information about the load sensitivity of a program through a static analysis, in the face of having no hope of maintaining accurate information about which paths will be taken through the program, or what objects will be produced or deallocated, the first approach to consider is phrasing the LSI computation problem as a global data flow analysis problem. Unfortunately, this problem does not fit nicely into a data flow analysis framework primarily because the computation involves essentially counting the amount of space requested and deallocated, which depends highly on the number of iterations that loops are executed, when these allocation and deallocation statements appear inside loops.

Thus, our algorithm is a recursive algorithm that traverses the control flow graph, until all edges of the infinite loop have been traversed. Only two instructions are of interest to the LSI computation problem, allocation (malloc) and deallocation (free) statements. At each point (i.e., statement) in the program, a value is assigned as the worst case LSI and the best case LSI for that program point. For the malloc, free, and other non-control program statements, the LSI value is computed as follows. In the following, LSI(i) is the LSI value at the program point just prior to instruction i, and LSI(i') is the LSI value at the program point just after instruction i.

```
LSI(i') = LSI(i) + s
where instruction i is malloc(s bytes)

LSI(i') = LSI(i) - s
where instruction i is free(s bytes)

LSI(i') = LSI(i)
otherwise
```

The other statements that have an effect on the LSI computation are join points in the program and loops. These computations are different for worst and best case analysis. At a join point j of path x and path y, the worst and best case LSI are computed as follows:

```
WORST_LSI(j) =
if WORST_LSI(x) > 0 or WORST_LSI(y) > 0, then
  WORST_LSI(j) = max(WORST_LSI(x),WORST_LSI(y))
if WORST_LSI(x) < 0 and WORST_LSI(y) < 0, then
  WORST_LSI(j) = min(WORST_LSI(x),WORST_LSI(y))

BEST_LSI(j) =
if BEST_LSI(x) > 0 or BEST_LSI(y) > 0, then
  BEST_LSI(j) = min(BEST_LSI(x),BEST_LSI(y))
if BEST_LSI(x) < 0 and BEST_LSI(y) < 0, then
  BEST_LSI(j) = max(BEST_LSI(x),BEST_LSI(y))
```

The full algorithm for computing the LSI of a procedure is provided in figure 5. Before this algorithm is executed we assume that the "dominating" while- or for-node which causes the program to execute continuously has been identified. We may simply bypass all nodes appearing before and after the dominating while- or for-loop node.

After the LSI has been computed for a program, the following rules are applied to classify the degree of load sensitivity of the program. The program is said to be:
(1) **MUST-BE load sensitive** if the worst case LSI > 0 and the best case LSI > 0.
(2) **MAY-BE load sensitive** if the worst case LSI > 0 and the best case LSI ≤ 0.
(3) **Load Insensitive** or **Load Irrelevent** if the worst case LSI ≤ 0 and the best case LSI ≤ 0

This set of rules is considered **conservative.** Some load insensitive programs may be classified as **MAY-BE** load sensitive. However, those programs identified as **MUST-BE** load sensitive are accurately identified.

4 Summary and Current Work

In this paper, we proposed a *structural* approach to performing load testing. A load sensitive index has been defined, and an algorithm was presented to find load sensitive paths, using the load sensitive index. This structural approach has several advantages over the functional approach to load testing: (1) It allows the load test to be performed when each module is developed. (2) It focuses on individual modules which are subject to load sensitive faults. (3) It can be automated.

This paper concentrates on our initial work on the structural approach to generating test data for load testing. Our next step is to determine the effectiveness of

ALGORITHM: Computing LSI.

INPUT: Annotated Control Flow Graph

OUTPUT: Load Sensitive Index and the classification of load sensitivity

METHOD:

1. Bypass all nodes until the outer while- or for-loop node is reached.
2. Call LSI(pointer, END-WHILE, worst_lsi, best_lsi);
3. The worst_lsi and the best_lsi are used to decide the classification of the program.

```
LSI(pointer, end-node-type, worst_lsi, best_lsi)
/* The pointer points to a node in the annotated control flow graph.node.   */
/* The end-node-type is the type of the node that the pointer points to.   */
/* The worst_lsi is the worst case LSI.                                     */
/* The best_lsi is the best case LSI.                                       */
{
    pointer = pointer->next;
    while ( pointer != NULL && pointer->type != end-node-type )
    {
        switch ( pointer->type )
        {
         case IF-NODE:
               handle_if(pointer, type, worst_lsi, best_lsi);
               break;
         case WHILE-NODE:
               LSI(pointer->next, END-WHILE, worst_lsi, best_lsi);
               Multiply the worst_lsi and the best_lsi by the number of iteration;
               break;
         case MALLOC-NODE:
               lsi = lsi + pointer->size; /*  The size allocated is incremented */
               worst_lsi = lsi;
               best_lsi = lsi;

               break;
         case FREE-NODE:
               lsi = lsi - pointer->size;/* The size freed decreses the LSI */
               worst_lsi = lsi;
               best_lsi = lsi;

               break;

         default:
               break;
         }
         pointer = pointer->next;
    }
}
```

```
handle_if( pointer, type, worst_lsi, best_lsi)
{
     int worst_lsi1, best_lsi1, worst_lsi2, best_lsi2;

     if (pointer->left != NULL)
            LSI(pointer->left, ENDIF, worst_lsi1, best_lsi1);
     if (pointer->right != NULL)
            LSI(pointer->right, ENDIF, worst_lsi2, best_lsi2);

     worst_lsi = max (worst_lsi1, worst_lsi2);

     best_lsi = min (best_lsi1, best_lsi2);

}
```

Figure 5: An Algorithm for Computing LSI

this approach in terms of eliminating load insensitive programs from the suite of programs initially pegged to be load tested. Based on these results, we will proceed to refine the static analysis. For example, we could attempt to identify infeasible paths and eliminate their consideration from the LSI computation. That is, by performing a constant propagation and folding phase before the LSI computation, we may find that certain branches will never be executed. Other methods are described in [6] and [4]. We are currently working on refining our conservative assumption for the number of executions of loops. Finally, our ultimate goal is the automatic generation of test data for programs that we identify as load sensitive, and then building a useful testing tool that incorporates these techniques.

References

[1] A. Avritzer and B. Larson. Load testing software using deterministic static testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 82–88, June 1993.

[2] A. Avritzer and E. Weyuker. Generating test suites for software load test. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 44–57, Seattle, Washington, Aug. 1994.

[3] A. Bertolino and M. Marrè. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, Dec. 1994.

[4] D.F.Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 48–54, Dec. 1989.

[5] D. Hamlet, B. Gifford, and B. Nikolik. Exploring dataflow testing of arrays. In *International Conference on Software Engineering*, pages 118–129, Baltimore, Maryland, 1993.

[6] D. Hedley and M. A. Hennell. The cause and effect of infeasible paths in computer programs. In *International Conference on Software Engineering*, pages 259–266, Imperial College, London, UK, 1985.

[7] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 73–86, Oct. 1991.

[8] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[9] R. E. Pather and J. P. Myers. The path prefix software testing strategy. *IEEE Transactions on Software Engineering*, 13(7):761–766, July 1987.

[10] S. Venkatesan and B. Dathan. Coverage measurement for path testing based on the concept of essential branches. *Journal of Information Processing*, 6(4):199–205, 1983.