HDD: Hierarchical Delta Debugging

Ghassan Misherghi Zhendong Su
Department of Computer Science
University of California, Davis
{ghassanm, su}@ucdavis.edu

ABSTRACT

Inputs causing a program to fail are usually large and often contain information irrelevant to the failure. It thus helps debugging to simplify program inputs. The Delta Debugging algorithm is a general technique applicable to minimizing all failure-inducing inputs for more effective debugging. In this paper, we present HDD, a simple but effective algorithm that significantly speeds up Delta Debugging and increases its output quality on tree structured inputs such as XML. Instead of treating the inputs as one flat atomic list, we apply Delta Debugging to the very structure of the data. In particular, we apply the original Delta Debugging algorithm to each level of a program's input, working from the coarsest to the finest levels. We are thus able to prune the large irrelevant portions of the input early. All the generated input configurations are syntactically valid, reducing the number of inconclusive configurations that need to be tested and accordingly the amount of time spent simplifying. We have implemented HDD and evaluated it on a number of real failure-inducing inputs from the `GCC` and Mozilla bugzilla databases. Our Hierarchical Delta Debugging algorithm produces simpler outputs and takes orders of magnitude fewer test cases than the original Delta Debugging algorithm. It is able to scale to inputs of considerable size that the original Delta Debugging algorithm cannot process in practice. We argue that HDD is an effective tool for automatic debugging of programs expecting structured inputs.

Categories and Subject Descriptors: D.2.5[Software Engineering]: Testing and Debugging—*Debugging aids, Testing tools.*

General Terms: Reliability.

Keywords: Automated debugging, Delta Debugging.

1. INTRODUCTION

Programmers spend a significant amount of their time debugging programs. Studies consistently show that software maintenance typically requires more time than any other programming activity [11]. For an exhibited bug, programmers must determine which portions of a given test case induce a program failure. This search phase of the debugging process is slow and arduous. Once the program's errant behavior is finally understood, the bug is often quickly fixed.

Many times, a programmer is given a large test case that produces a failure. Reducing this test case simplifies the debugging process because there are fewer irrelevant details contained within the test case, allowing the programmer to focus on issues pertinent to the failure. Minimizing test cases has traditionally been left to humans.

1.1 Delta Debugging

Delta Debugging, a technique by Zeller and Hildebrandt, is an approach for automating test case minimization [20]. It consists of two algorithms:

Simplification: In this algorithm, the failure-inducing input is simplified by examining smaller configurations of the input. The algorithm recurses on the smaller failure configurations until it cannot produce a smaller configuration that still produces a failure; and

Isolation: This algorithm attempts to find a passing configuration such that with the addition of some element it becomes a failing configuration. The algorithm works in both directions, finding bigger passing cases that are subsets of a failing case.

Isolation produces outputs which are less intuitive as a debugging aid for the programmer. The single element difference is not individually responsible for the failure, it merely guarantees that some symptom is exhibited. The programmer then has to sift through the potentially large failure-inducing configuration to determine what else may be responsible for the failure. Although isolation may generally be faster than simplification, for large test cases, it may lead to worse running times because of the time spent testing the large configurations. In this paper, we will be mainly concerned with the first algorithm, simplification. More details on Delta Debugging are given in Section 3.1.

1.2 Hierarchical Delta Debugging

Input data is often structured hierarchically; however, Delta Debugging ignores input structure and may attempt many spurious input configurations. Our insight is that the existing input structure can be exploited to generate fewer input configurations and simpler test cases for more effective automated debugging. In this paper, we propose a *Hierarchical Delta Debugging* algorithm, HDD, to validate our hypothesis.

There are numerous examples of input data with recursive definitions. When input is nested and at least partially balanced, there is temptation to take advantage of this in our test case minimization algorithm. We present several examples of input data for which Hierarchical Delta Debugging is applicable. In the most general case,

ICSE'06, May 20–28, 2006, Shanghai, China.

any data defined by a context-free grammar is a good candidate for Hierarchical Delta Debugging. If a context-free grammar is necessary for the definition of a particular language, a simple regular language cannot suffice. The data set is thus likely to be nested, giving us an advantage over standard Delta Debugging. We give below a few concrete scenarios where Hierarchical Delta Debugging may be applied:

Programming Languages: Programs can make use of a Hierarchical Delta Debugger when considered as input into a compiler or interpreter. If a large program causes a failure in a specific compiler, the Hierarchical Delta Debugger can operate over the program's abstract syntax tree (AST). The minimum configuration is found at all levels of the AST. For example, the algorithm first finds the minimum configuration of classes, prototypes, global variables, and top level functions. It then recurses into methods, statements, and local declarations. Following this, it determines the minimum configuration of sub-statements and expressions. This process is performed to the lowest level of the AST. Declarations demonstrate that there are cases when some nodes depend on higher non-ancestral nodes within the AST. We will later present an approach to solve this problem.

HTML/XML: HTML and XML are also excellent candidates for Hierarchical Delta debugging. These languages, as generated both by humans and machines, are widely deployed. In both cases the inputs tend to be well nested. XML can benefit much from a syntactically aware algorithm because it adheres to a strict grammar. Furthermore, XML is meant to be general, thus a single implementation may be applicable to many program inputs.

Video Codecs (compressor/decompresser): Hierarchical Delta Debugging can be applied successfully to cases when data is of limited depth. Consider a simple video codec that crashes on a particular video sequence due to a subset of the frames not necessarily immediately neighboring one another. Suppose that the encoding scheme consists of several groups containing a single key frame and multiple delta frames. Our algorithm first selects the minimum configuration of groups in the failing sequence. It then recurses one level below to determine which delta frames are inducing the failure. Reaching the result is significantly faster than a flat implementation since we have removed the irrelevant groups first before considering their individual frames.

UI Interactions: User interfaces also demonstrate a potential use for Hierarchical Delta Debugging. GUI applications may crash after complicated user sessions. The programmer benefits from a minimized session in that the failure can be reproduced and examined in a simpler form. UI interactions can be categorized into levels of tasks, *e.g.*, "open the document" and "print." These high level tasks can be composed of other tasks such as "opening the print dialogue," "selecting the printer," "configuring the printer," and "selecting the pages." At the lowest level of this task hierarchy are the actual actions required to perform these tasks, including mouse movements and other events. With this hierarchy, one can apply Delta Debugging to find the minimum number of interactions that induce some error. Unfortunately, automating the creation of this hierarchy is difficult; a user must categorize various actions manually, perhaps inside the interaction recording program.

```
void f()
{
    int x;  int y;
    if (x!=0) { y= x; } else { return 0; }
    while (y!=0) { y--; }
    return y;
}
```

Figure 1: An example program.

1.3 Main Contributions

In this paper, we develop a general Hierarchical Delta Debugging algorithm to exploit hierarchical characteristics of program inputs. When attempting to find which portions of an input to prune, there is little reason to choose arbitrary points for division. Instead, we work along the boundaries of the tree from the top to the bottom. By limiting ourselves to one level at a time, we are examining smaller groups of nodes for minimization. This also exploits the relative independence of nodes on different branches of the tree.

Our algorithm produces only syntactically valid configurations provided that the input specific tree manipulators are capable. While this seems counter to the purpose of failure finding, it is indeed consistent. The goal of the algorithm is to determine a failure-inducing input that should be valid to the program. Parsing related issues are not the problem addressed by this paper. If our input is not well-formed, techniques such as Delta Debugging should be employed to determine why the parser does not fail gracefully. In many cases, spurious test configurations that fail at the parse level before triggering our desired bug waste testing time.

The original Delta Debugging algorithm produces test cases that can be difficult to read by minimizing things such as identifiers, and removing section boundaries. This is often counter to the purpose of making the debugging process easier. Our approach can make the minimized input more closely resemble the original structure. A developer may find this easier to understand because of its similarity to a real test case.

We have implemented our algorithm and applied it in two settings: (1) bugs in the `GCC` compiler, and (2) bugs in the Mozilla web browser for XML processing. On real failure-inducing inputs from the `GCC` and the Mozilla bugzilla databases, our algorithm generates orders of magnitude fewer test cases and produces simpler outputs than the original Delta Debugging algorithm, resulting in significantly faster minimization time. This evaluation confirms the effectiveness of our Hierarchical Delta Debugging algorithm when applied to programs accepting structured inputs.

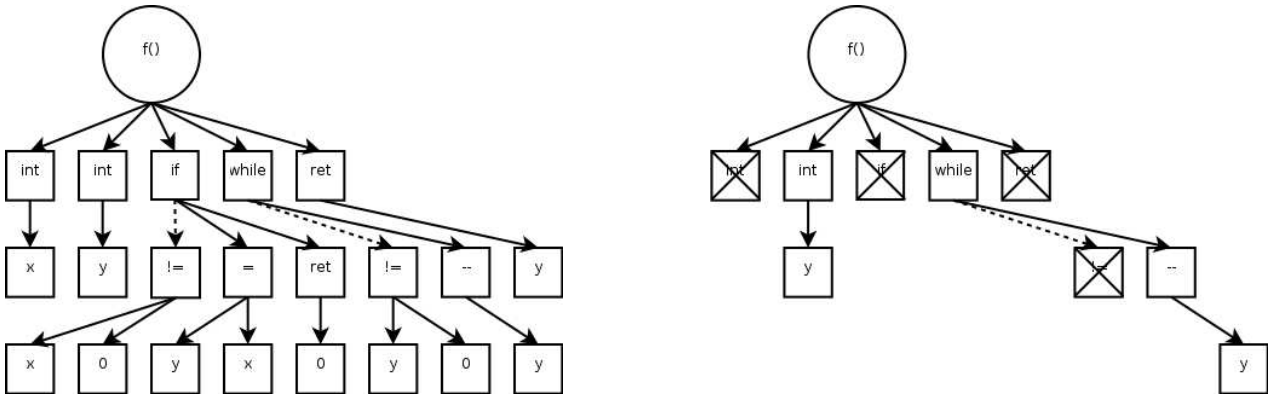
1.4 Paper Outline

The rest of the paper is structured as follows. We first use a concrete example to explain the intuition of our Hierarchical Delta Debugging algorithm (Section 2). Next, we present our algorithm (Section 3), followed by an empirical evaluation to compare our technique to the original Delta Debugging algorithm (Section 4). Then, we survey closely related work (Section 5). Finally, we discuss possible improvements to our current algorithm (Section 6), and conclude (Section 7).

2. EXAMPLE

In this section, we demonstrate a contrived C program to illustrate our algorithm. Details of the algorithm will be given in Section 3.

The first step of our algorithm is to parse a failure-inducing input. Using the parse tree, we manipulate the input and generate new test cases. Consider the contrived program shown in Figure 1. The



(a) Figure 1's AST.

(b) The result of our algorithm on Figure 2a.

Figure 2: Applying Hierarchical Delta Debugging to the code in Figure 1.

parsed input should be the AST of the program. Figure 2a shows the AST of the program in Figure 1. In this example, there is only one function causing the compiler to fail. Let us assume that the post-decrement operator, "`y--`," is the source of the failure.

Our algorithm begins processing at the top level of the parse tree. We first determine the minimum configuration of functions, global variables, prototypes, and type definitions. There is only one top level form in our example, the function `f()`. Our algorithm will first try to exclude this function from the configuration. The function is required to induce the compiler error; it will thus be included in the top level's minimum configuration.

Once the minimum configuration of a particular level is determined, we try the next level of the AST. Note that there may be multiple parents for all the nodes at a level. We use the standard Delta Debugging algorithm [20] to determine the minimum configuration of all these nodes. Each subtree could instead be treated individually, yielding an algorithm similar to a pre-order tree traversal. We examine this optimization in more detail in Section 6.

Figure 2b shows the result of the Hierarchical Delta Debugging algorithm applied to the original AST in Figure 2a. The minimized AST corresponds to the following code:

```
f() {  int y;   while (0) { y--; }  }
```

We stated previously that the post-decrement operator was the source of the compiler failure. Note that the minimum nodes relevant to this error are chosen for any level. For example, if the variable `y` were not declared at the second level, our program would not type check and thus the compiler may not exhibit the failure. The `while` statement on the second level is of particular interest. The condition of the while statement was deemed irrelevant. Handling this case correctly is left to the tree manipulator; ours inserted an empty condition, `0`, in-place of the original condition. Some implementations may attempt to replace the loop with its body. There are several nodes in the AST that have required children. To produce syntactically valid inputs, we must handle removal of their children on a case by case basis, most likely by substituting in the smallest allowable syntactual fragment.

3. HIERARCHICAL DELTA DEBUGGING

3.1 Background on Delta Debugging

Before presenting HDD in detail, we explain the simplifying Delta Debugging algorithm [20], hereafter called ddmin, as it is integral to HDD. The input to the ddmin algorithm is a failure-inducing configuration, *i.e.*, a list of elements that causes a program to fail when given as input. The goal of the ddmin algorithm is to determine a subset of the input such that no one element can be removed from it while preserving the failure. Zeller and Hildebrandt call this a *1-minimal test case* [20]. The ddmin algorithm proceeds by executing the following steps to find a 1-minimal test case:

1. *Reduce to subset:* Split the current configuration into n partitions. Test each partition for failure. If a partition does induce the failure, then treat it as the current configuration and resume at Step 1.
2. *Reduce to complement:* Test the complement of each partition. If any induces the failure then treat it as the current configuration and resume at Step 1.
3. *Increase granularity:* Try splitting the current configuration into smaller partitions, $2n$ if possible, where n is the current number of partitions. Resume at Step 1 with the smaller partitions. If the configuration cannot be broken down into smaller partitions, the current configuration is 1-minimal and the algorithm terminates.

Each split is chosen to produce sub-partitions of similar size to remove as much as possible from the input. This characteristic of the ddmin algorithm makes it much like a binary search. If each split is successful, we can reduce the input by at least half each iteration (assuming "reduce to subset"). Unfortunately this is not likely to occur because the input is not split according to the structure of the failure-inducing configuration. Furthermore, the failure-inducing portions of the file may also be scattered throughout the file. Our approach addresses these issues better when simplifying structured inputs.

3.2 Algorithm Description

We are concerned with hierarchical inputs. Why not first determine the minimal failure-inducing configuration of coarse objects? Following this intuition, we can recursively minimize configurations starting from the top-most level. By limiting simplification to one level of the hierarchy at a time, we can prune data more intelligently. At each level we must try multiple configurations to determine the minimum failure-inducing configuration. This process employs the ddmin algorithm at each level.

Before a level is processed, we must first know how many nodes there are in a level, and name each node. Without addressing nodes

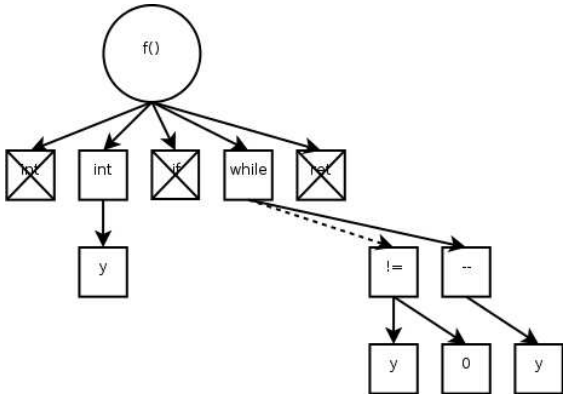


Figure 3: The AST after the first iteration of HDD.

we cannot compactly select a configuration for testing. Node naming is implemented by traversing the tree and assigning a unique identifier to each node at the level of interest. Nodes are named only before we process their levels.

Unparsing a configuration of the tree is required for each attempted test configuration. This is the most frequent operation executed by HDD. Our implementation traverses the tree, checking for node inclusion in the configuration before printing a particular node.

After determining the minimum configuration at a level, one approach to refining the input is to output the new configuration and re-parse it for further processing. This approach, though simple, is not ideal since we spend time re-parsing needlessly. We eliminate this step by providing primitives for tree pruning. This is implemented by removing all irrelevant nodes from a level. Following tree pruning we proceed to the next level. Figure 3 shows the first iteration of the Delta Debugging algorithm applied to the AST in Figure 2a.

Algorithm 1 shows HDD's pseudocode. Line 2 begins the process at the top level of the tree (where $level = 0$). We count and tag the nodes at the current level on lines 3 and 8 with an auxiliary function TAGNODES. If there are nodes at the current level (line 4), we try minimizing the nodes of that level using the standard Delta Debugging algorithm, DDMIN (line 5). Although not shown in the algorithm, ddmin must call a testing procedure to determine if a configuration induces the failure. This procedure must be given the input tree and the level currently under inspection. Line 6 removes the irrelevant nodes from the tree with the auxiliary function PRUNE. On line 7, we progress to the next level of the tree for processing.

3.3 Algorithm Complexity

We now discuss HDD's complexity with respect to the number of tests performed. Our HDD algorithm has the same worst-case complexity as ddmin, generating quadratic number of tests in the size of the original input. Our worst-case happens when the input is a flat list, essentially reducing HDD to one call to ddmin on the entire input.

For inputs with nested structures such as programs, we expect HDD to perform much better. We empirically validated this claim using a few real-world examples, and we delay that discussion to Section 4. In this section, we instead examine the performance of HDD with specific input characteristics. We consider the following two scenarios:

The Ideal Case: Suppose HDD is run on a balanced tree of size n with constant branching factor b such that for each parent *exactly one* child remains in the configuration. There are $\log_b n$ levels in this tree. At each level, we invoke ddmin, and thus require $O(b^2)$ tests. Consequently, we run $O(b^2 \log_b n)$ tests, or simply $O(\log n)$, because b is constant.

Algorithm 1 The Hierarchical Delta Debugging Algorithm

```
1:  procedure HDD(input_tree)
2:      level ← 0
3:      nodes ← TAGNODES(input_tree, level)
4:      while nodes ≠ ∅ do
5:          minconfig ← DDMIN(nodes)
6:          PRUNE(input_tree, level, minconfig)
7:          level ← level + 1
8:          nodes ← TAGNODES(input_tree, level)
9:      end while
10: end procedure
```

A More Realistic Case: The above scenario is too idealistic for many cases. Instead let us suppose that exactly m children are relevant from each relevant parent. While this is unlikely to occur in practice, we may consider m to be an upper bound on the number of children that are taken in all cases. We cannot choose more than b nodes from every parent; consequently we must have $m \leq b$. Intuitively, it is clear that the number of nodes to examine becomes fairly large further down the tree.

At the top level we examine the root of the tree. For any subsequent level, i, we examine b nodes for every included parent, or $b\left(m^{i-1}\right)$ nodes. We run ddmin at each level, therefore we try at most $O\left((b\left(m^{i-1}\right))^2\right)$ tests at level i. Summing over the entire tree, we have:

$$
\begin{aligned}
& 1 + \sum_{i=1}^{log_b n} \left(b * m^{i-1}\right)^2 \\
= \;& 1 + b^2 \sum_{i=1}^{log_b n} \left(m^{i-1}\right)^2 \\
\leq \;& 1 + b^2 \left(\sum_{i=1}^{log_b n} m^{i-1}\right)^2 \\
= \;& 1 + b^2 \left(\frac{m^{log_b n} - 1}{m - 1}\right)^2, \text{when } m \neq 1.
\end{aligned}
$$

Given m and b, we conclude that HDD runs worst-case:

$$O\left(\left(\frac{m^{log_b n} - 1}{m - 1}\right)^2\right) = O\left(m^{2log_b n}\right)$$

number of tests, when $m \neq 1$. Substituting 1 for m in the initial summation above, we have $1 + b^2 log_b n$. Thus, we run $O(log_b n)$ tests, which is consistent with the idealistic example above. If we choose all b nodes from each parent, *i.e.*, when $m = b$, we have $O(b^{2log_b n}) = O(n^2)$. Thus, we run absolute worst-case $O(n^2)$ tests. For a specific case in between, such as $b = 4$ and $m = 2$, we have $O(2^{2log_4 n}) = O(n)$.

Though the relative benefit gained by HDD depends on the shape of the input tree, we never perform asymptotically more tests than the original Delta Debugging algorithm. If the tree is well balanced, we can expect large portions of the original input to drop whenever

we remove a node high in the tree. This property is what enables us to achieve better asymptotic bounds.

3.4 On Minimality

We now examine the problem of simplifying failure-inducing inputs from a more formal perspective. In particular, we investigate the quality of produced output from a simplifying algorithm. The most natural metric for a generated output is its size, and the obvious notion to consider is *minimality*: How small is the output? Zeller and Hildebrandt propose various definitions for minimality in the context of ddmin [20]. We introduce similar definitions for trees instead of sequences of elements as used in their definitions. First, we define the meaning of tree simplification in terms of two predicates $simplify$ and $simplify*$.

DEFINITION 3.1 (SIMPLIFY). *For any two trees T and T', the predicate $simplify(T, T')$ holds iff T' can be derived from T by removing a single node.*

DEFINITION 3.2 (SIMPLIFY*). *For any two trees T and T', the predicate $simplify*(T, T')$ holds iff T' can be derived from T by removing zero or more nodes, and more precisely:*

- $simplify*(T, T)$*; and*
- $simplify*(T, T') =$

$$\exists T'' \left(simplify(T, T'') \wedge simplify*(T'', T') \right)$$

DEFINITION 3.3 (GLOBAL-TREE-MINIMAL). *Given program P and input tree T, T' is* global-tree-minimal *if $simplify*(T, T')$ and $P(T') =$ fail, and for all T'' such that $simplify*(T, T'')$ and $P(T'') =$ fail, it holds $|T'| \leq |T''|$.*

Ideally, we would want an algorithm that finds a *global-tree-minimal* input that induces the failure. Unfortunately, we will show next that this is infeasible in general. Our notion of *global-tree-minimality* coincides with Zeller and Hildebrandt's notion of global-minimality, because a tree is merely a hierarchy of the actual input configuration. *Global-tree-minimality* is a very difficult problem computationally; we show that the problem is NP-complete. First, we formulate the decision version of this problem, GMT, where we treat the given program P as a constant-time oracle because we are interested in the number of generated test configurations.

DEFINITION 3.4 (GMT). *Given program P, a failure-inducing input T, and a positive integer K, is there a tree T' with $|T'| \leq K$, such that $simplify*(T, T')$ and $P(T') =$ fail?*

THEOREM 3.5. *GMT is NP-complete.*

PROOF. GMT is clearly in NP, because given a GMT instance (P, T, K), we can guess a configuration S from T, and verify both $|S| \leq K$ and $P(S) =$ fail, all in polynomial time.

To show that GMT is NP-hard, we reduce the hitting set problem (HS) to GMT. Recall that in the hitting set problem, we are given a collection C of subsets of a finite set S and positive integer $K \leq |S|$, and the question is: Is there a subset $S' \subseteq S$ with $|S'| \leq K$ such that S' contains at least one element from each subset in C? The hitting set problem is known to be NP-complete [8].

Given an instance (C, S, K) of HS, we construct an instance (P, T, K) of GMT as follows:

- $P(t) =$ fail iff for each $c \in C$, there exists a child s of t such that $s \in c$; and
- $T = \mathsf{root}(s_1, \ldots, s_n)$, where $\mathsf{root}(s_1, \ldots, s_n)$ denotes a tree with root as the root and $s_i \in S$ as its children.

This is clearly a polynomial time reduction. It is also straightforward to verify that (C, S, K) has a hitting set S' with $|S'| \leq K$ if and only if (P, T, K) has a failure-inducing input T' with $|T'| \leq K$. Here we measure the size of a tree as the number of children under the root node "root." □

Global-minimality is thus a difficult problem; neither HDD nor ddmin can claim to produce global-minimal configurations. Instead of attempting to achieve such an elusive goal, we will attain minimality with respect to the immediate "neighbors" of a failing configuration. With such a goal, we can merely examine all neighbors of the current failing configuration until none induce the error. Then we have attained a local-minimal input. Program failures tend to mirror this: Some portion of the input is relevant to the failure, the rest can be removed piece by piece.

DEFINITION 3.6 (1-TREE-MINIMAL). *Given a program P and input tree T, T' is* 1-tree-minimal *if* simplify*(T, T') *and $P(T')$ produces a failure, and for all S such that* simplify(T', S)*, $P(S) \neq$ fail.*

Before evaluating HDD's output with respect to such a property, let us discuss minimality with ddmin. DDmin's behavior at the finest granularity is what allows it to assure 1-minimality: it tries all individual elements of the configuration alone, or tries to remove single elements from the current configuration until no single element can be removed. By definition, 1-minimality is assured since the algorithm terminated and thus there is no single element that can be removed. Our approach constrains ddmin to subsets of the failure-inducing configuration. When we call ddmin to remove nodes at a specific level, we may enable nodes at some other level of the tree to be removed.

Our algorithm will not re-attempt to remove nodes on levels higher than the current level, hence HDD may not always produce 1-minimal configurations. Any algorithm which does not produce a 1-minimal configuration does not produce a *1-tree-minimal* configuration either, since the elements in the configuration must also be nodes in the tree. We now demonstrate several alternate algorithms derived from HDD that satisfy 1-minimality or *1-tree-minimality*.

Suppose all internal nodes of the input tree represent only collections of other nodes and do not contribute any of the actual elements that makeup the the program's failure-inducing configuration. Consider non-terminals from context-free grammars as a good example of this. If we ensure that all leaf nodes are put on the last level, then HDD's final call to ddmin will include all elements in the configuration. This guarantees that the final result is 1-minimal with respect to these individual elements, by merely relying on ddmin to do so. By considering trees of this type, HDD alone is sufficient for 1-minimality. This approach does not adequately take advantage of the hierarchical nature of HDD: the final output may not be *1-tree-minimal*. For example, consider a C program inducing some failure in a C compiler. Assume that the compound statement "{}" is inside the aforementioned program and is irrelevant to inducing the failure. Removing just one of the contained braces causes a parse error, potentially preventing the compiler from inducing the failure. Removing the entire statement from the program may still induce the failure. If a tree has a single parent node for the compound statement, then a *1-tree-minimal* input cannot contain the compound statement.

We now present an algorithm, HDD+, consisting of a greedy phase and a final *1-tree-minimality* assurance phase. First we perform the greedy phase: a call to HDD on the input configuration tree. This phase will attempt to trim the tree as best it can without consideration for *1-tree-minimality*. The second optimality oriented phase is symmetric with the final step of ddmin. It will try

to remove individual nodes from the tree one by one in a BFS-like manner. It will repeat the attempt on the entire tree continually as long as the previous iteration successfully removed at least a single node. This algorithm produces a *1-tree-minimal* configuration by definition: the final input tree does not have a single node that can be removed, otherwise the algorithm would not have terminated. In the worst case, the entire algorithm generates $O(n^2)$ number of test inputs on trees of size n. As we have previously shown, HDD generates worst-case $O(n^2)$ test inputs. Each iteration of the second phase requires at most n tests, and since it removes at least one element from the tree each iteration, it cannot iterate more than n times. It follows that the second phase also generates worst-case $O(n^2)$ test inputs, affirming our analysis.

We present another algorithm for *1-tree-minimality*, HDD*. This algorithm repeatedly calls HDD until a single call fails to remove a single element from the input tree. Since ddmin will attempt to remove every node in the tree individually at least once, HDD will attempt to remove every node in the tree individually at least once. If HDD cannot remove a single node from the tree, the tree is *1-tree-minimal*. It is possible that each iteration of this algorithm removes only one node from the tree, and since each iteration generates worst-case $O(n^2)$ test inputs, it follows that HDD* generates worst-case $O(n^3)$ test inputs. We find this is very unlikely in practice. Implementing HDD* is simple with an already working HDD implementation, so for comparison we included it in our empirical results. We hope by our evaluation to demonstrate two conclusions: (1) worst-case analysis for HDD, ddmin, and HDD* does not reflect what happens in practice, and (2) 1-minimality is not necessarily the best criteria for input minimization.

PROPOSITION 3.7. *Both HDD+ and HDD* produce* 1-tree-minimal *configurations.*

We suspect HDD and HDD* may produce output close to the global minimum for practical settings. We leave this as future work to validate this claim.

4. EMPIRICAL EVALUATION

We have evaluated HDD in two settings: bugs in the `GCC` compiler and bugs in XML processing of the Mozilla web browser. In this section, we discuss our experience with the algorithm.

4.1 The C Programming Language

We created a tree processing module for the C programming language in order to test various `GCC` failure-inducing programs. Our AST manipulator was implemented as an extension to Elsa [13], a GLR parser for C++. We augmented Elsa's AST nodes with methods and data to facilitate manipulation. There was little inheritance among AST nodes in the Elsa framework, forcing us to duplicate functionality more than we should have. For example, an `if` statement has specific variables for each of its three children: the condition, the `then` branch, and the `else` branch. A `for` loop, on the other hand, has entirely different variables for its children: an `initializer`, a `condition`, a `post-incrementor`, and a `body`. Although the process of printing and pruning is similar for both of these statements, their implementation could not be adequately factored. This lack of generality made implementation tedious. We examine alternative approaches in Section 7.

We now apply HDD to a concrete example. To simplify comparison, we choose the same program demonstrated in Zeller and Hildebrandt's work on delta debugging [20]. Figure 4 shows a program that causes `GCC-2.95.2` to fatally abort, even though the poorly written program is valid C code. The problem lies within the `for` loop of the function `mult()`, where an incorrect floating point optimization causes `GCC` to crash.

```
double mult( double z[], int n )
{
    int i;
    int j;
    for (j= 0; j< n; j++) {
        i= i+j+1;
        z[i]=z[i]*(z[0]+0);
    }
    return z[n];
}

int copy(double to[], double from[], int count)
{
    int n= (count+7)/8;
    switch (count%8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while (--n > 0);
    return (int)mult(to,2);
}

int main( int argc, char *argv[] )
{
    double x[20], y[20];
    double *px= x;

    while (px < x + 20)
        *px++ = (px-x)*(20+1.0);

    return copy(y,x,20);
}
```

Figure 4: A program that crashes GCC-2.95.2.

The Hierarchical Delta Debugging algorithm first determines that the only relevant function is `mult()` by calling ddmin on the first level of the code's AST (Figure 5a). Next, it removes the return type, `double`, and the `return` statement because they are also irrelevant for the bug (Figure 5b). At the next level, the algorithm determines that the loop initializer, condition, and post-incrementor are not necessary to induce the failure (Figure 5c). The algorithm then unsuccessfully attempts to simplify the expressions inside the two assignment statements. It cannot produce a smaller configuration from these expressions because they are integral to inducing the failure, and HDD terminates with the program shown in Figure 5c. Reaching this output required 86 test cases, significantly fewer than the 680 tests performed by ddmin.

To further test the performance of HDD, we applied it to other real programs from the `GCC` bugzilla database (`http://gcc.gnu.org/bugzilla`). Our examples exhibited a number of characteristics, each highlighting interesting results. We selected the first three examples based only on our ability to reproduce failures within our limited testing environment. Table 1 shows our empirical results. For each program in the table, we list size, bug report number from the bugzilla database, and the number of tests and final sizes as

```
double mult(double *z, int n)
{
  int i;
  int j;
  for (j=0;j<n;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
  return z[n];
}
```

(a) The minimized first level.

```
mult(double *z, int n)
{
  int i;
  int j;
  for (j=0;j<n;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

(b) The minimized second level.

```
mult(double *z, int n)
{
  int i;
  int j;
  for (;;) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

(c) The final output.

Figure 5: HDD applied on various levels of the code in Figure 4.

computed by ddmin, HDD, and HDD*. We use a token as our unit of size because a token abstracts details such as identifier length and whitespace. Our algorithm could have minimized such characteristics, but doing so is unlikely to help the developer. We now discuss the result for each test case:

bug.c: The first program, bug.c, is the same example we demonstrated in Figure 4. HDD ran almost an order of magnitude fewer tests than ddmin—a common result in our evaluation. HDD's output size was nearly identical to that of ddmin, though HDD's is slightly smaller. The output from HDD is shown in Figure 5c. For comparison, we show ddmin's output below (formatted for easier reading):

```
t(double z[], int) {
  int i;
  int j;
  for(;;j++) {
    i=i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

In this example, the main distinction was the removal of the loop terminator, "`j++,`" something ddmin failed to do. In ddmin's case, there are two possible ways to remove the expression "`j++`" from the loop without modifying the loop itself. One possibility is for the post-increment operator to fall on the correct boundary when ddmin operates at the granularity of two characters. The other is for the entire expression to fall on a correct boundary at the granularity of three or more characters, assuming the introduction of whitespace. Indeed, ddmin's output is very sensitive to whitespace. Although only slightly better, HDD's output could have been shortened if our implementation had attempted to remove parameters from function signature. A quick comparison with ddmin's output shows that the parameter is not necessary, though it failed to remove the parameter altogether:

```
t(double z[],int) { ...
```

It is interesting to note that ddmin was able to remove the parameter's name only because of `GCC`'s lax treatment of invalid function definitions.

boom7.c: The program boom7.c is relatively small in size and contains many variable declarations with one deep expression that induces the bug. HDD ran more than an order of magnitude fewer tests than ddmin, and the output was also significantly smaller—about half the size of ddmin's output. Early versions of our implementation failed to minimize the program significantly, but as we added support for minimizing unary expressions properly, HDD was able to produce a small failure inducing statement. The output from ddmin is less informative; it is not clear which portions correspond to the original. The output is not even parseable at points, for example:

```
(long int)(signed short)
    (var0=9>(var1=var1=8))""""
```

Compiling ddmin's output under a newer version of `GCC` produces a syntax error.

HDD* minimized the file even further than HDD. On the first iteration, HDD dramatically reduced an expression crucial for the failure, leaving many variables without references in the program. Since the original variable references were at deeper levels than the variable declarations themselves, attempting to remove them in the first iteration caused a type-checking error that prevented the compiler from failing. The second iteration was then able to remove the irrelevant variables since they were no longer needed.

cache.c: Program cache.c is relatively large, mainly due to the high number of header files included by the preprocessor. This characteristic is less than ideal for our approach since the resulting AST is flat, with many function prototypes and a few relevant functions following. The ddmin algorithm was quite capable of scaling with respect to the program size since most of the input is not pertinent. Even so, HDD still ran more than an order of magnitude fewer tests with an output size just under that of ddmin's. The most significant contributor was HDD's ability to unravel a heavily nested and parenthetical expression. Had our implementation removed parameters from function signatures or semi-colons after case statements, it would have removed five additional tokens. Not surprisingly, HDD* was able to remove a single unreferenced variable, yielding a slightly smaller program than HDD.

cache-min.c: The synthetic example, cache-min.c, exhibits a characteristic that affirms our approach. The program is a nearly minimized version of cache.c with parentheses introduced in all subexpressions. It may seem strange that a nearly minimized input requires nearly the same number of test cases

File	size (tokens)	bug report (id)	ddmin tests (# of tests)	HDD tests (# of tests)	HDD* tests (# of tests)	ddmin size (tokens)	HDD size (tokens)	HDD* size (tokens)
bug.c	277	unknown [20]	680	86	164	53	51	51
boom7.c	420	663	3727	144	304	102	57	19
cache.c	25011	1060	1743	191	327	62	61	58
cache-min.c	145	1060	1074	114	182	71	59	59

Table 1: Experimental results. All tests were performed against GCC version 2.9.5.2.

for the ddmin algorithm, but a nearly minimized program actually induces poor behavior in ddmin. This is because the algorithm tries in vain to remove large portions of the program. Although HDD is also limited by this, HDD constrains the number of nodes examined at one time. If a tree is fairly well balanced and of significant depth, we scale quite well in comparison. The output from cache-min.c was slightly more minimized than cache.c because all references to the variable "`line`" were dropped and thus HDD was able to eliminate the declaration "`int line;`". This is the same declaration that previously allowed HDD* to minimize cache.c, beyond what HDD was able to do.

4.2 XML

Unlike C, XML seems intuitively to be an interesting application for HDD because, unlike C, it is general, very strict, and often deeply nested. These characteristics make XML an ideal candidate for HDD. In this section, we experimentally confirm this.

Because of XML's generality, a single implementation of HDD can be applied uniformly in many application domains. The process of our experimentation is evidence of this: we first implemented the XML tree processing module for HDD and then searched for XML document types on which to experiment. Our implementation is very simple; it is written in less than 150 lines of Python code using the DOM API. Although it may not always produce documents that are valid with respect to a specific document type definition (DTD), the generated documents are always well-formed.

While compilers and web browsers may tolerate bad inputs, XML parsers do not. This strictness is beneficial, even when users create XML documents directly, because it forces users to conform before facing mysterious errors or vendor lock-in. Because XML parsers are more strict, ddmin is more likely to produce documents that fail to parse, and is thus less likely to succeed. Consider that all tags present in a document must also have their closing tags to be processed. Removing one tag without the other will not induce the failure, even if the tags are both irrelevant. Furthermore, tags themselves must have matched angle brackets. Any configuration that produces a boundary covering an odd number of angle brackets will be frivolous.

Extensible Stylesheet Language Transformation (XSLT) is an XML document type that is used to transform other XML documents into another form. XSLT has a notorious history of inducing bugs; as such it was a good starting point for our bug search. The documents in Table 2 are the first XML documents with which we were able to reproduce errors in Mozilla. The Mozilla bugzilla database (`http://bugzilla.mozilla.org/`) gives such an example. It contains a bug entry for ms-tour.xsl (id 248258), a complex transformation that attempts to generate a knights tour of a chess board. The Mozilla web browser crashes with a segmentation fault upon processing this document. The file is comprised of 16500 characters and 433 lines of XML. Table 2 summarizes our results. In particular, we performed the following tests on this file:

ddmin: The ddmin algorithm failed to minimize this program. At larger granularities, ddmin did not remove significant portions of the document aside from those inside large comments. At smaller granularities, ddmin's cache of tested configurations ended up exhausting memory on a system with 4GB of RAM. With the cache disabled, ddmin retests duplicate configurations. Based on ddmin's diagnostic output, it was clear after one day of testing that we had not completed a significant percentage of total testing. We concluded that ddmin could not simplify this data under normal circumstances.

ddmin-line: To simplify the matter, we ended up minimizing the document on a line-by-line basis instead. Tags were generally isolated on individual lines. Accordingly, all possible boundaries respect angle bracket openings and closings, though they may not respect open and close tags. By treating lines or tags atomically, ddmin will not process tag attributes, but since we evaluate ddmin's performance by the number of lines it produces, this is not necessary.

As expected, ddmin had trouble significantly reducing the file. The output was 92 lines long and took 1092 tests to produce. On examination, the output seems unsatisfactory: it is still quite complex and includes many consecutive open and close tags that are irrelevant to inducing the failure. It may seem strange that ddmin was unable to remove them, but one must remember that the output is indeed 1-minimal since removing any one tag will not produce a parseable document. These tag pairs were not removed because they either contained tags that were not removed until later granularities, or they were on an odd-numbered line.

HDD and HDD*: Table 2 shows that HDD was more successful than ddmin in minimizing this input file. The output is only 8 lines and took 129 tests to produce. We observe again that HDD takes an order of magnitude fewer tests, and manages to simplify the failure-inducing configuration to a concise one. The difference in output size for ddmin and HDD is very significant, with a factor of ten. Repeated application of HDD did not yield a smaller configuration. Still, HDD* did not incur a significant cost in terms of number of additional test cases.

We found another file that is small enough for ddmin to process at the character level. The XSLT file uiwrapperauto.xsl proved small enough for a direct comparison. Since ddmin was able to simplify the file successfully, we separated the tags from ddmin's output onto separate lines, to facilitate comparison. Our experiment showed that ddmin executed a factor of 30 more tests than ddmin-line, and nearly equivalent output. It was, however, able to shorten a few of the actual tag attributes. HDD also attempted to exclude attributes, though ddmin-line did not. Because this file is smaller, the overall results of this experiment are less drastic than ms-tour.xsl.

File	size (lines)	bug (id)	ddmin (# tests)	ddmin-line (# tests)	HDD (# tests)	HDD* (# tests)	ddmin (lines)	ddmin-line (lines)	HDD (lines)	HDD* (lines)
ms-tour.xsl	433	248258	failed	1092	124	167	failed	92	8	8
uiwrapperauto.xsl	66	207358	5757	277	105	143	46	43	15	15

Table 2: Experimental results for the XML study.

Almost paradoxically, we are unable to say HDD produces 1-minimal inputs, but HDD in our experience seems to be more effective than ddmin. The reason for HDD's success is quite simple: HDD can intelligently handle input languages which are context-free, while ddmin is suited for regular languages.

5. RELATED WORK

The most relevant work to our own is Zeller's seminal work on Delta Debugging [19, 20]. Zeller hints at intelligent partition choices in [19]. The idea would significantly reduce the number of test cases run by the algorithm, though it falls short of realizing our motivational insight. Partition boundaries should be chosen with respect to a specific granularity of the input. It follows intuitively that we should start with the coarsest granularity and move towards the finest. Zeller and Hildebrandt briefly mention applying Delta Debugging to specific domains including a short reference to context-free grammars [20]. Sterling and Olsson's recent work on "program chipping" [14] is also related, and addresses a similar problem for Java. It uses simple tree manipulation techniques to produce test cases, while ours is based directly on Delta Debugging. The relative strength of both approaches is yet to be explored.

Rather than examining the input to a program, program slicing [15, 16] can be used to isolate relevant portions of a program that are necessary to yield some result. Program slicing can be performed either statically or dynamically (with respect to one concrete run) [2]. These techniques ease debugging by removing irrelevant portions of the failing program [1, 17]. It is worth noting that both Delta Debugging and program slicing can be used cooperatively. By first minimizing the input, we may significantly simplify the program slice and trace. These two techniques are not necessarily competitors; they can be complementary.

PSE [12] is a static analysis technique for diagnosing program failures. It can be viewed as a program slicing technique, however it is more precise because of its consideration of error conditions. A motivating example is dereferencing a NULL value. It is similar to Das's earlier work, ESP, a symbolic dataflow analysis engine [6].

Bug isolation is related to simplifying failure-inducing input. Rather than focusing on minimizing the input, it focuses on finding the cause underlying the failure. Delta Debugging the program state space is used as the mechanism for this technique [5, 19]. The algorithm attempts to locate the state differences between passing and failing runs. This determines the relevant variables and values that infect the program to failure. A similar technique is applied to multi-threaded applications [4]. Instead of focusing on state, the technique examines the thread schedule differences of passing and failing runs.

Whalley's work on isolating failure-inducing optimizations is also related to our work [18]. His approach automatically isolates errors in the *vpo* compiler system. The search is performed on the sequence of optimizations performed by *vpo* to find the *first improving* optimization that causes incorrect output. The approach is fairly domain specific.

Liblit *et al.* use a sampling technique to reduce the runtime overhead of collecting successful and failing runs [9]. They also propose to use statistical learning techniques to infer the failures from many sampled runs [9, 10].

Work in error explanation for static analysis relates to our approach. Many tools produce error traces when a program violates its specification. However, understanding the error and locating the cause is usually left to the user. Several techniques have been developed to address this problem. Ball *et al.* suggest an approach to localize error causes [3]. Their idea is to find transitions in the error trace that do not appear in correct traces. Groce and Visser suggest a different approach for the same problem [7]. Given an error trace, they compute other error traces leading to the same assertion violation to compare with traces preserving the assertion.

6. FUTURE WORK

In this section, we discuss limitations on our current implementation and interesting directions for improving the algorithm.

First, our Elsa extension is limited in a number of ways. Certain list types are immutable, hampering simplification of expressions such as arguments to functions. When multiple variables are declared in the same declaration statement, Elsa splits the declaration into multiple statements.

Our approach works best if there are few dependencies between data at different levels of the input. An excessive amount of dependence yields output that is not minimized adequately. The program input domain and minimization requirements should be evaluated to determine if HDD* should be used.

Another limitation of our work is that the programmer must provide infrastructure for tree processing. This infrastructure is responsible for parsing the input, unparsing a configuration, and pruning nodes from the input tree. This turned out to be non-trivial, even when implementing over the Elsa C/C++ parser. XML has a simpler syntax, and though it is more general, implementation was easier.

Ideally a Hierarchical Delta Debugging implementation would automate tree processing by using only a context-free grammar that describes the input syntax. From the CFG, a parser would be generated that directly creates the input tree as well as the rudimentary tree processing routines. There are already several tools that generate AST-like tree builders from context-free grammars. With these, we could then provide generic implementations for tree manipulation. Tagging, printing, and pruning are very similar in all cases. Each node would contain one list of its children for simplicity, treating each symmetrically. This approach still presents a few problems.

Context-free grammars use recursion to deal with lists of data. If the generated tree mirrored this exactly, the tree would not be ideal for our algorithm. This kind of unbalanced trees would increase the number of iterations that our algorithm takes. We could solve this problem by augmenting the context-free grammar with simple annotations for each non-terminal that we desire to be flattened. The tree builder would simply flatten all non-terminals of this type, so that the the entire list remains on the same level.

Another problem with automated tree processing is that the syntactic validity of the output from a particular configuration cannot always be guaranteed when given an arbitrary context-free gram-

mar. As a simple example, consider arithmetic expressions from the C programming language. The addition expression can be represented as an expression followed by the terminal '+' followed by another expression. If either the right or left subexpression is removed, the terminal '+' must also be removed. For simplicity, we could treat the '+' operator like any other removable node in the tree. The Delta Debugging algorithm would thus attempt configurations with or without the '+' operator, sometimes producing syntactically invalid test-cases.

From a practical point of view, producing some invalid configurations is not a significant loss considering the added generality. We may perform more tests than absolutely necessary, but we still gain over the original Delta Debugger from trying configurations in a hierarchical manner. This algorithm would remove subtrees from the input as early as possible, much like our original algorithm.

Finally, for some input types, executing ddmin over all nodes at the same level is not necessary. If this is the case, we may be able to call ddmin on only the children of a parent. By making the individual calls to ddmin smaller, we limit its worst-case behavior. The algorithm would mirror a breadth-first search.

7. CONCLUSIONS

In this paper, we have presented HDD, a Hierarchical Delta Debugging algorithm that exploits input structure to minimize failure-inducing inputs. We have evaluated the algorithm on some real-world examples. Our empirical evaluation confirms that Hierarchical Delta Debugging can reduce the number of generated tests and at the same time produce smaller output than the original Delta debugging algorithm. Although some simplicity is lost, if projects have many non-trivial bug reports, the required work to implement the necessary tree manipulating routines is worthwhile. In these scenarios, input minimization without structural knowledge of the data does not scale. Here are a few interesting directions for future work: (1) We plan to examine alternate techniques of optimizing simplification by exploiting node independence. One possible approach is the aforementioned branch containment technique to limit ddmin to children of a particular parent node. We believe this yields significant speedups for certain inputs; and (2) it is also interesting to provide a general context-free grammar framework to facilitate the adoption of this technique.

Acknowledgments

We are grateful to Prem Devanbu for his suggestions on conducting an evaluation and his feedback on drafts of this paper, and to Earl Barr, Christian Bird, Ron Olsson, Chad Stirling, Gary Wassermann, and Andreas Zeller for feedback on drafts of this paper and for their encouragement of this work. We would also like to thank the ICSE anonymous reviewers for their valuable comments.

8. REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[3] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, 2003.

[4] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220, 2002.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, May 2005. to appear.

[6] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[7] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN 2003*, pages 121–135, 2003.

[8] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, NY, 1972.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.

[11] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[12] R. Manevich, M. Sridharan, and S. Adams. PSE: explaining program failures via postmortem static analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 63–72, 2004.

[13] S. McPeak. Elsa: The Elkhound-based C/C++ parser. `http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/index.html`.

[14] C. Sterling and R. A. Olsson. Automated bug isolation via program chipping. In *Sixth International Symposium on Automated Debugging and Analysis-Deiven Debugging (AADEBUG'05)*, 2005. To appear.

[15] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[16] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[17] M. Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[18] D. B. Whalley. Automatic isolation of compiler errors. *ACM Trans. Program. Lang. Syst.*, 16(5):1648–1659, 1994.

[19] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, 2002.

[20] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Engineering*, 28(2), Febuary 2002.