Iterative Delta Debugging

Cyrille Artho

Research Center for Information Security (RCIS), AIST, Tokyo, Japan

Abstract. Automated debugging attempts to locate the reason for a failure. Delta debugging minimizes the difference between two inputs, where one input is processed correctly while the other input causes a failure, using a series of test runs to determine the outcome of applied changes. Delta debugging is applicable to inputs or to the program itself, as long as a correct version of the program exists. However, complex errors are often masked by other program defects, making it impossible to obtain a correct version of the program through delta debugging in such cases. Iterative delta debugging extends delta debugging and removes series of defects step by step, until the final unresolved defect alone is isolated. The method is automated and managed to localize a bug in some real-life examples.

1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [8,10]. Automated test runs allow efficient verification of software as it evolves. However, in the case of failure, defect localization (debugging) is still a largely manual task.

In the last few years, several automated debugging techniques have been developed in order to facilitate fault-finding in complex software [5,13]. Delta debugging (DD) is one such technique [7,14]. It takes two sets of inputs, one which yields a correct result and another one which causes a failure. DD minimizes their difference while preserving the successful test outcome. DD can be applied to the program input or the source code of the program itself, using two different revisions of the same program. The latter variant treats the source code as an input to DD. It therefore includes a compile-time step, which produces mutations of the program to be analyzed, and a run-time step, where the outcome of that mutation is verified by testing.

DD obtains an explanation for a test failure. The scenario considered here is where a test fails for the current version. If an older correct version exists, DD can be used to distill the essential change that makes the new version fail on a given test, and thus reduce a large change set to a minimal one [14]. DD is applicable when there exists a known version that passes the test. For newly discovered defects, this may not be the case. For such cases, we propose iterative delta debugging (IDD). The idea is based on the premise that there exists an old version that passes the test in question, but older versions of the program may have other defects introduced earlier that prevent them from doing so. By successively back-porting fixes to these earlier defects, one may eventually obtain a version that is capable of executing the test in question correctly [1].

IDD yields a set of changes, going back to previous revisions. The final change, applied to the oldest version after removal of earlier defects, constitutes an explanation for the newest defect. Furthermore, the same algorithm that is used to back-port fixes to

older versions can also serve to port the found bug fix forward to the latest revision (or a version of choice). Our approach is fully automated. We have applied the algorithm to several large, complex real-world examples in different programming languages and types of repositories. Even though it was not known a priori whether a working revision could be found, we have found working revisions in some cases.

This paper is organized as follows: Section 2 introduces the idea behind IDD. DD and IDD are described in Sections 3 and 4, respectively. The implementation and experiments carried out are described in Sections 5 and 6. Section 7 concludes, and Section 8 outlines future work.

2 Intuition Behind IDD

Developers have used change sets for debugging before. IDD automates this process:

1. A test fails on the current version. It is assumed to have passed on an older version.
2. By successively going back to older versions, one tries to obtain a version that passes the test.
3. One tries to distill a minimal difference between the "good" and the "bad" version, which constitutes the change that introduced the defect.

Step 2 tries to isolate two successive revisions, one that passes a test, and another one that fails. Step 3 attempts to minimize the difference between these two revisions. In that step, we assume that delta debugging (DD) is used to minimize a given change set while preserving the given test outcome [14].

If a version passing the test (a "good" version) is known a priori, then the search for the latest defective version can be optimized by using binary search instead of linear search. This idea has been implemented in the source code management tool `git`, which is used to maintain the Linux kernel sources [11].

The process becomes more complex when the "good" version is not known. Assume there exists a test that fails on a current version. We will call this outcome *fail*. A correct result is denoted by *pass*. Besides these two outcomes, there may also be different incorrect outcomes of the test, denoted by *err*. A set of changes between two versions of a program is referred to as a *patch.*

A test case may not be applicable to older revisions due to missing features or other defects that prevent the test from executing properly. In this case, IDD utilizes DD to apply the necessary changes from a newer version to an older version, to allow a test to run. Figure 1 shows how IDD builds on DD. IDD starts from a version that fails (version 4 in Figure 1). Unlike in the original scenario for DD, a version that passes is not known a priori. IDD successively goes back to previous versions and assesses whether the same failure persists. If the test outcome differs, DD is applied to the last failing version and the older version. This identifies the source code change that made the old version behave differently. One IDD iteration thus either (a) skips a version where the outcome does not change, (b) finds a correct version, or (c) eliminates a older defect (*err*) that covered a newer one.

In the example in Figure 1, version 3 produces case (c): It does not pass the test, and even fails to reproduce the behavior of version 4. DD is then applied between versions 3

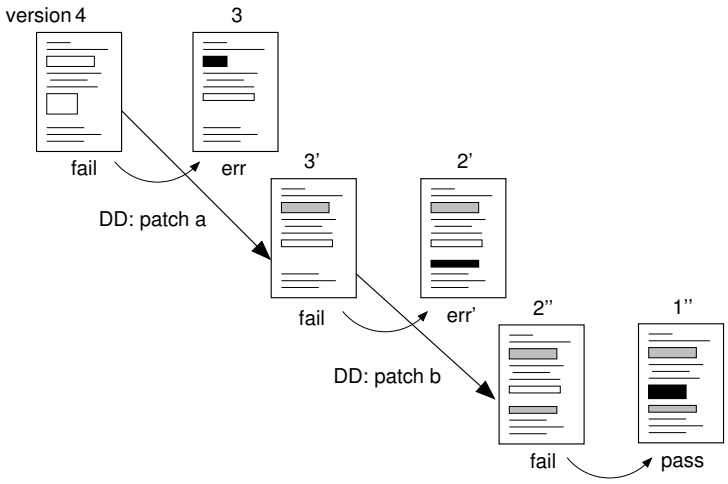

Fig. 1. Iterative delta debugging.

```
prev_version = current_version = latest_version();
patch = {}
original_result = current_result = test(current_version);
while (current_result ≠ pass) {
   current_version = predecessor(current_version);
   current_result = test(current_version ⊕ patch);
   if (current_result ≠ original_result) {
      patch = DD(current_version, original_version);
   }
   prev_version = current_version;
}
return patch;
```

Fig. 2. IDD algorithm in pseudo code.

and 4. The resulting minimal change, patch a, can then be applied as a patch to version 3, fixing the earlier defect in version 3, and producing a new version $3'$. The resulting version fails again in the same way as version 4 did. In Figure 1, the change set (patch) that is back-ported is shown by a thick arrow pointing to the correct, changed code.

The iterative process of IDD continues by applying this patch to older version (such as version 2) before running tests. This produces version $2'$, on which the test is run. That version behaves differently from $3'$, so DD is applied again to find the minimal change required to fix the program. This produces a new patch b (the change between $3'$ and 2). This patch usually contains changes of the previous patch a. The resulting new version is therefore called $2''$. After version 2 is repaired, IDD continues. The patched version 1, $1''$, passes, and IDD terminates successfully in this example.

IDD will eventually find a version that passes the test, or run out of older versions (see Figure 2). IDD is fully automatic as long as the test process does not require human

Step	Change set	Test verdict
1	00000000	fail
2	11110000	fail
3	11111100	pass
4	11111000	fail
5	11110100	pass
6	00000100	pass

Fig. 3. Delta debugging illustrated.

intervention. The process starts from a version where the test in question successfully compiles, and produces a known, well-defined output. (Even though that output is erroneous, it is important to use it as a measure against changes.) IDD successively tries older revisions, and uses DD as a subroutine upon failure. DD produces a minimal change set from a newer (working but not fully correct) revision will be back-ported to the older one, which produces an error.

3 Delta Debugging

3.1 Delta debugging

Delta debugging (DD) uses bisection to minimize the difference between two versions of program input while preserving the behavior that the first input generates. It can also be applied to the program source code itself. When applied to program source code, DD typically operates on a description of the difference between two revisions. The difference set generated by the Unix tool `diff` provides this change set readily. It also has the advantage that variations of the change set can automatically be applied by `patch`, followed by compilation and execution of the program.

Delta debugging applies bisection to an entire change set. This evaluates changes in contiguous subsets only, and ignores vast parts of possible subsets of a change set. Since the size of a power set is exponential in the size of the original set, exhaustive evaluation of all possible change sets is impossible. In the following discussion, a "successful" or passing test is a test whose outcome corresponds to the desired value, such as the same kind of test failure observed in a previous revision.

Figure 3 illustrates DD on a simple example. The change set includes eight elements, the presence or absence of each is illustrates by a "1" or "0", respectively. Let the first element by the leftmost bit in the change set, and the eighth element be the rightmost bit. Initially, the empty change set is tried. The test, when executed unchanged, fails. DD then attempts to isolate the reason of the failure. The assumption is that if the test passes with only the first half of the change set being active, then the reason for the failure must lie in the first half of the change set. Conversely, if the test fails with the second half of the change set disabled, then at least a part of the second half of the change set is necessary for the test to succeed. This is what happened in the example in Figure 3: The test still fails after the initial bisection of the state space. Therefore, at least one element of the last four elements in the change sets is necessary to achieve the desired outcome.

In the next DD step (3), the subset of the last four elements is again bisected, and the last two elements are ignored. As the test passes, they can be safely dropped from the change set. Now, the algorithm backtracks and analyzes the subset containing elements 5 and 6. A direct implementation of backtracking would again analyze the change set of step 2. As that one is known to fail, it can be skipped. The change set in step 4 therefore contains elements 1–5; the test fails, confirming that element 6 is necessary. Step 5 confirms that element 5 can be dropped; further backtracking analyzes the first half of the change set, which again does not contribute to the test outcome in this example.

DD assumes that each element of a change set is independent of all others. As the format of the Unix `diff` tool is line-based, DD works best on data where one line is independent of any other line, such as a flat configuration file. Obviously, program source code does not conform to this property. For local code changes, though, DD provides a good approximation of the minimal change set. DD fails in cases where the hierarchical structure of program source code conflicts with line-by-line analysis of changes. In typical programming languages, a class scope surrounds declarations of functions and methods, which contain declarations of local variables and code. Higher-level statements cannot be removed without removing everything within its scope. Therefore, when trying to eliminate the addition of a new method, DD tends to eliminate statements and declarations but not necessarily the entire method scope. Conversely, when eliminating the removal of particular statements, it is not possible to remove an entire method without removing all the program statements it contains.

Figure 4 shows some of the problems arising with DD. It shows two examples in "unified diff" format, as produced by `diff -u`, with file names and line numbers omitted. On the left hand side, a C function is added to the code. DD will analyze the given changes in conjunction with many other changes, spanning hundreds of lines. Bisection of the state space may happen at any location, and may not coincide with function or method boundaries.

Incomplete functions will not compile, resulting in invalid code subsets whenever the syntactic structure of the target language is violated. Because of this, DD can only manage to remove unused functions if bisection hits the boundaries of its declaration. In Figure 4, for DD to succeed fully, bisection needs to hit the function boundaries and the scope of the preprocessor `#if`/`#endif` construct. In other cases, DD only manages to remove the code inside the function, and, if it works from bottom to top, the declaration of local variable `x`. On the right hand side, where a function is removed, the situation is even worse. A reduction of the change set would correspond to adding code.

Function addition	Function removal
`+ #if 0`	`- static void`
`+ static int`	`- bar(int y) {`
`+ foo (void * data) {`	`-   int x;`
`+   int x;`	`-   x = y;`
`+   x = 42;`	`-   if (y)`
`+ }`	`-     bar(--x);`
`+ #endif`	`- }`

Fig. 4. Different use cases for delta debugging.

As statements cannot be compiled without an enclosing function declaration, bisection needs to hit the function boundary, or none of the changes regarding statements can be removed. This example shows why DD is of limited usefulness if large parts of the code, especially function declarations and interfaces, change.

3.2 Hierarchical delta debugging

The problem of having to conform to a syntactic structure also occurs when applying DD to hierarchical data, such as XML. Previous work has implemented hierarchical delta debugging to produce correctly nested changes of XML documents [7]. In program code, it is very difficult to obtaining change sets that include their hierarchical scope. Luckily, the changes produced by the `diff` utility contain some hierarchy in it: Change sets consist of changes to individual files, which are in turn broken up into so-called "hunks", which contain a number of line-based changes. Figure 5 illustrates this hierarchy using the "unified diff" format. In real code repositories, some changes are entirely local in the scope of a single file or a block of code where it occurs. Extending DD to include the hierarchy of the generated patches can therefore improve precision of DD in these cases, as shown in the experiments in Section 6.

This hierarchy provides possible boundaries for the state space search. Instead of bisecting the state space in the middle, based on the number of lines involved, bisection proceeds hierarchically, across files, hunks, and lines. First, sets of changes across files as a whole are analyzed. If a change for an entire file cannot be ignored, a more fine-grained search proceeds on hunk level, then on line level.

Usage of the patch file hierarchy improves recognition of local code changes, but it does not take care of interdependent changes. Addition or removal of a function may be taken care of by multiple iterations of DD: Once all calls and references to a function are removed, then the function itself may be removed as well. Unfortunately, even hierarchical DD cannot deal with certain changes affecting multiple files. For example, in cases where the signature of a function changes, its definition and all instances of its usage have to be changed simultaneously. A bisection-based algorithm such as DD cannot isolate such changes and produces overly large change sets.

Patch	Hierarchy
`Index: file1`	File
`@@ -42,2 +42,3 @@`	Hunk
`  context`	
`+ addition`	Line
`  context`	
`@@ -84,4 +85,3 @@`	Hunk
`  context`	
`- removal`	Line
`+ addition`	Line
`- removal`	Line
`  context`	

Fig. 5. The hierarchy of a change set produced by `diff -u`.

4 Iterative Delta Debugging

IDD starts with a test failure in a new revision. The goal is to find an older revision that passes the given test. Whenever the outcome of a test changes in a different way than succeeding, e. g., by executing an infinite loop, delta debugging is used to create a minimal patch that preserves the former outcome.[1] Iteration proceeds until no older revisions are available, a revision where the test passes is found, or a timeout is reached.

4.1 Iteration strategy

A given test case (or a set of test cases) is compiled, executed, and evaluated by a shell script. This evaluation is successively applied to older revisions, until a working version of the code is found. In most cases, the outcome of a particular test does not change from one revision to the next. Instead, older revisions contain changes affecting other parts of the program. Delta debugging is only necessary when the test case of interest changes.

Previous work implemented this iteration strategy but involved manual patch creation [1]. Usage of DD automates our method. Unlike in previous work, patches are not accumulated, but replaced with a new patch each time delta debugging (DD) is invoked.

4.2 Forward porting

So far, the given algorithm attempts to locate a correct version of the program by going back to older revisions. Patches (minimized by DD) are applied whenever a test cannot be executed in an older version. If the algorithm is successful, it will eventually find a revision where the given test passes. The final change set will contain a bug fix for the given test, as well as all the "infrastructure" needed to execute it. If the bug fix applies to a version that is much older than when the test was implemented, the portion of the change set containing new features that are necessary for a given test may be large. Intermediate design changes may have taken place, making it impossible to directly apply the resulting patch to the current version. However, the primary concern is usually to fix the latest revision of the software rather than an old one. Therefore, after having found a successful revision, IDD is again applied in reverse, going from the correct version to the current one. DD is again invoked as a subroutine whenever a patch cannot be applied. In this way, forward IDD generates a patch for the current version.

5 Implementation

5.1 Compilation and test setup

Within each iteration of IDD, and for evaluating changes of a test, IDD uses a set of scripts to automate program compilation and testing. In detail, this involves the following steps:

[1] Even though nested dependencies require multiple DD iterations to be removed, only one iteration of DD is run at each time except at the final step, when a fix-point iteration is performed.

1. Update of the source code to a new revision. This step may fail due to unavailability of the source repository (e. g. due to a network outage) or because an older version is unavailable.
2. Patching the source code. Patching is a likely point of failure, as any major changes in the source code, including formatting changes, make it impossible to apply a patch to a version other than the one the patch was generated for.
3. Configuration, preceded by deletion of all compiled files. Configuration may require re-generating a build file. Deletion of all compiled files is necessary, as the patching process may create files that do not compile. This would result in the object file of the older version (which successfully compiled) being used in a current test run, and falsify test results.
4. Compilation. Also referred to as the build process, this step generates the executable program and may fail because the given version cannot be compiled before or after application of a patch. If a given version does not compile even when unpatched, then it can be skipped.[2]
5. Test execution. Upon successful compilation, given test cases are executed. Test execution failure may be detected by a given test harness, e. g. if a known correct output is not generated by the test case. However, it also frequently happens that tests fail "outside" the given test harness. There are two ways for this to happen:
   (a) Catastrophic failure: Programming errors, such as incorrect memory accesses, may lead to failures that cannot be caught by the test harness. Therefore, the test harness of the application has to be complemented by an "outer" test harness that is capable of detecting such failures.
   (b) Incomplete test output verification: A given revision may not contain the full code to verify the test output. In that case, such a revision may erroneously report a failed test as successful. The outer test harness has to double-check that the lack of a reported failure actually implies success.[3]

Any failure in the steps above is treated as a critical failure that requires the current version to be fixed. The only exception is if a test fails in exactly the same way as a previous revision, in which case it is assumed that the test outcome has not changed. Otherwise, any change in the result of test execution requires invocation of delta debugging.

5.2 Test evaluation accuracy

It should be noted that IDD cannot guarantee that a correct revision (which passes a given test case) is detected as such. If DD removes code that does not contribute to a test failure, but is vital for a test to pass, then IDD cannot recognize a future successful test run as such anymore, as the functionality for the test to pass has been removed by DD. Figure 6 illustrates the problem on an example that could be run in C or in Java.

[2] It sometimes occurs that a repository contains a version that cannot be compiled, for instance, because a developer committed only a subset of all necessary changes.

[3] This may be difficult to achieve in practice, because a given output may not contain enough information to judge whether a modified version still achieves the intended result when a test case passes. In our case, we have also monitored the contents of log files and the amount of memory consumed by tests in order to determine whether source code changes produced behavioral changes in a test inadvertedly.

```
void test1() {
  int result;
  result = 0;
  result = runTest();
  assert(result != 0);
}
```

Fig. 6. Example showing a potential code removal that would prevent a test from passing.

Let us assume that the test fails in the current revision, i. e., `runTest` returns 0. This fact is preserved when line 4 that calls `runTest` is removed. Unfortunately, this removes the entire test functionality from the program. A different revision where `runTest` returns 1 would not be recognized as being correct. Even worse, if the code is compiled as a C program, the initialization of `result` and the `return` statement may be removed as well. This causes the return value of the test function to be undefined. During execution, the value probably corresponds to the contents of an element of a previous stack frame that occupied the same memory, but such behavior is platform-dependent. The test harness has to be augmented in order to prevent such deviations.

In order to maintain the exact behavior of failing tests, static or dynamic slicing [3,13] could be used. Slicing would ensure that DD does not remove any lines that contribute to the value of the test result. However, slicing tools are not portable across programming languages, and do not scale well to large programs. We have therefore chosen a less stringent approach. In addition to monitoring the output as closely as possible (using an external test harness), memory consumption and time usage are also checked for deviations from the expected previous value. Furthermore, usage of uninitialized data in C programs is prevented by inspecting compiler warnings and the final change set generated by DD. This process is currently not fully automated, but could be automated by using memory checking tools such as valgrind [9].

5.3 Implementation architecture

The iterative step of IDD, which applies a given patch across several revisions, is implemented as a shell script using the compilation and test setup described above as a subroutine. The script iterates through existing revisions until a change in behavior is detected. After that, it stops, and control is transferred to the DD program.

When used on a subversion code repository, IDD can take advantage of the fact that all versions are globally and consecutively numbered. Stepping through older revisions is therefore trivial. When using CVS, though, global revision numbers are not available. They were recovered by pre-processing the code repository. During this step, a revision counter was assigned to each revision that was more than five minutes apart from the next one. Revisions being less than five minutes apart were regarded as a single version that was committed using multiple CVS invocations.

DD is implemented as a Java program that takes a patch set derived by the Unix `diff` tool as input. It parses the patch file and produces an internal representation of the state space of all possible patch sets. It then iterates over the state space of all possible change sets. Each change set is produced as a modified patch, which is applied

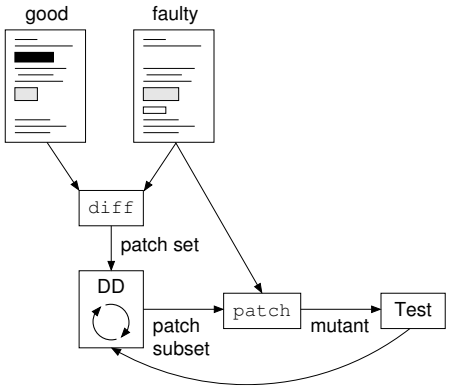

Fig. 7. Implementation architecture of DD.

to the faulty version using the `patch` tool. The resulting version reflects a subset of all changes between the good and the faulty version. Compilation and testing are then used to decide whether the mutated version still produces the desired outcome. DD continues its state space search based on this outcome, until the state space is exhausted.

Usage of the Unix `diff` and `patch` tools eliminated the need for a custom representation of the difference between two program versions. This made the DD algorithm rather simple, except for the cache structure, which was difficult to implement for hierarchical DD. While the recursion scheme is elegant to implement, caching adds a subtle interaction between recursion hierarchies of completed, partially completed, and incomplete parts of the search space. To ensure that the state space is bisected correctly even in the case of a nested hierarchy, about 400 unit tests were used. These tests covered many corner cases, but two additional incorrect corner cases were found by inspecting the output when the tool was run.

6 Experiments

Table 1 lists the three example applications taken for the experiments. The projects were chosen according to the following criteria:

- The project had to be sufficiently large to warrant automatic debugging, and have a certain maturity to contain enough older versions to be available for testing.
- The full history of all revisions had to be available.
- Tests had to be fully automated and repeatable, including any configuration files needed to run the test.
- The bugs in question had to be sufficiently well documented to be verifiable, and interesting enough to warrant attention. (This typically meant that the bugs in question had not been resolved for weeks or months.)

For each experiment, the outcome using IDD with "classical" delta debugging and hierarchical delta debugging are shown. Hierarchical delta debugging always fared better than its non-hierarchical counterpart, although the differences vary.

Table 1. Overview of projects used for experiments.

Project name	Description	Implementation language	Repository	
			Size [KLOC]	System
Uncrustify	Source code formatter	C++	20	Subversion
Java PathFinder	Java model checker	Java	70	Subversion
JNuke	Java VM/run-time analyzer	C	130	CVS

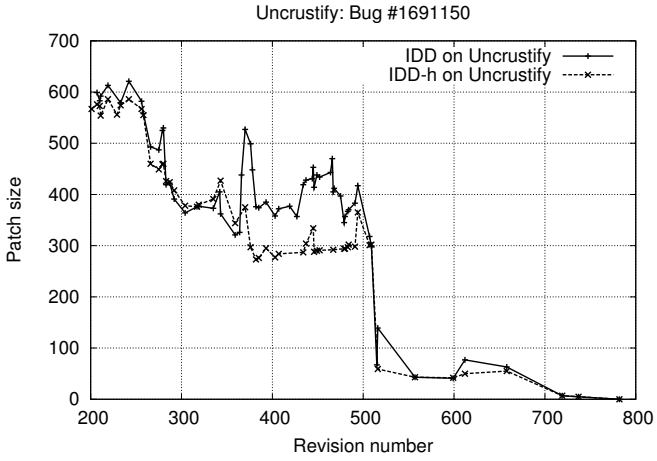

Fig. 8. Result of using IDD on Uncrustify.

6.1 Uncrustify

Uncrustify is a source code formatting tool for various programming languages including C, C++, and Java. In addition to formatting code, Uncrustify is also capable of changing the code itself. For instance, one-line statements following an `if` statement do not have to be surrounded by curly braces. Uncrustify can be used to add these optional braces, facilitating future changes. As a case study, bug number 1691150 in the sourceforge bug tracker was used. This bug describes how C++–style one-line comments (starting with `//`) are sometimes incorrectly moved to the wrong code block when they are converted to C-style comments (enclosed by `/*` and `*/`) while optional curly braces are added for `if` and `while` statements.

Figure 8 shows the result of running IDD on that case. The sudden growth of the change set at revision 494 corresponds to a refactoring where the large number of command line options was specified differently in source code. DD could not eliminate this refactoring. As older versions supported fewer and fewer options, the expected growth in the patch set was sometimes compensated by the shrinkage of the part of the patch that concerned command-line options.

IDD could not find a version passing the given test. Unfortunately, revisions 200 and older could not be checked out with the given subversion (`svn`) client. We assume

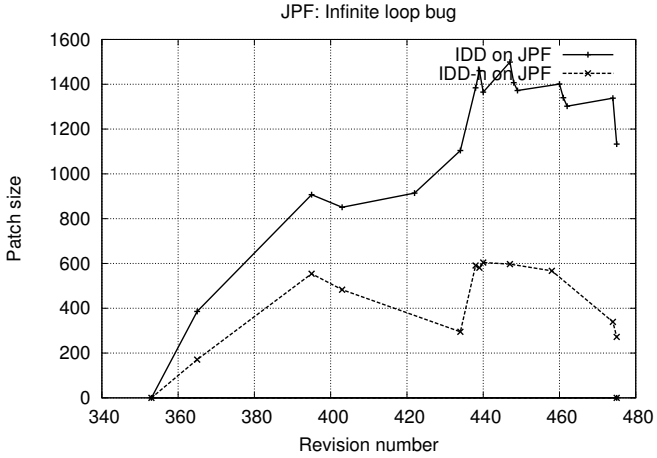

Fig. 9. Result of using IDD on Java PathFinder.

that the server-side repository was converted to a later version at that stage, making it impossible to retrieve old revisions.

6.2 Java PathFinder

Java PathFinder (JPF) is a Java model checker that analyzes concurrent Java programs for assertion violations, deadlocks, and other failures [12]. In version 4, a complex input is not processed correctly. The defect is manifest in an incorrect result after several minutes of processing: The analysis report of JPF states that a faulty input (a Java program containing a known defect) is correct. Because the point of failure is extremely difficult to locate, this problem has persisted for more than a year and has not been resolved yet. In fact, it was this case study that triggered the work of this paper.

A much older version of JPF, 3.0a, which is not maintained in the same repository, produces a correct result. However, using that version for identifying a change set would not be useful, because the entire architecture of JPF has been redesigned since version 3.0a was released. Hence, IDD was applied to different revisions of the source repository containing all revisions of version 4. The goal was to find a revision in the new repository that could pass the test. Perhaps the bug was introduced after the architectural changes that took place prior to the migration to a public source repository. In that case, IDD could find it.

We have applied IDD using revision 475 of version 4 as a starting point. IDD iterated through older revisions up to version 353, which passed the test (see Figure 9). In this case, no changes had to be back-ported to that revision. Therefore, the visible part of the graph shows the size of the patch fixing the defect. Delta debugging was unable to identify a very small change set to fix the newer, defective versions. As a result of this, the initial patch of 386 and 171 lines, respectively, grew successively during back-porting. Sometimes, a later iteration of DD took advantage of previously

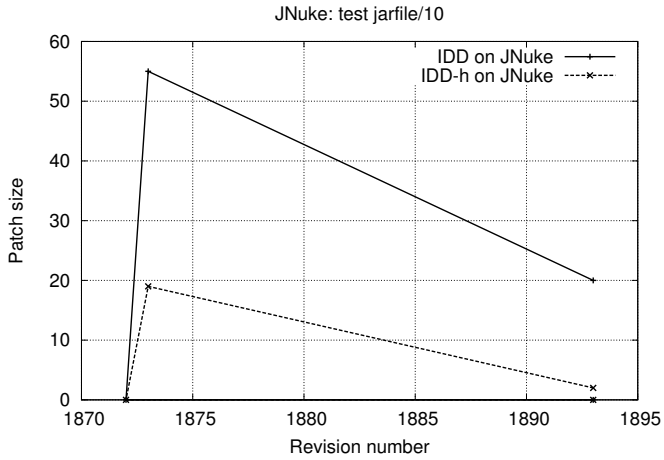

Fig. 10. Result of using IDD on JNuke.

resolved dependencies and shrank the patch again. However, at the end, an unwieldy patch of more than 1,000 lines remained for standard IDD, and a patch of 272 lines for hierarchical IDD.

This is still too large for the patch being meaningful to a human programmer. The generated change set still includes refactorings amidst functional changes, and while it fixes the given bug, it breaks other features of JPF. Because of this, we did not follow up with this bug fix further to actually repair JPF.

6.3 JNuke

In JNuke, IDD was applied to find the reason of a memory access problem in the Jar file reader under Mac OS X 10.4 that was not found under Linux. This is maybe the most typical case in which IDD can be applied: Code that is often tested on one platform (in this case, Linux) but rarely on another one (Mac OS). Such tests may pass on the main development and test system, but fail on a different platform. As the system is periodically tested on the other platforms, some known good versions exist, but regression defects may go undetected for some time.

As Figure 10 shows, IDD was very successful here. The test could be run without any adaptations on older versions, until it passed at revision 1872. The graph shows the size of the patch fixing the problem. Hierarchical IDD even found a minimal patch of just two lines, identifying the fault precisely. This example is a case where IDD would have found the test quickly enough to be useful for replacing human effort in debugging.

6.4 Summary

Table 2 summarizes the outcome of all experiments. As can be seen, the hierarchical version of IDD always produces significantly smaller patches, although the difference varies. In either case, the number of DD cycles within DD would have been

Table 2. Summary of experiments.

Project name	Size of final patch set		Number of DD invocations		Number of build/test runs	
	IDD	IDD-h	IDD	IDD-h	IDD	IDD-h
Uncrustify	666	567	62	57	52483	39170
Java PathFinder	1133	272	18	13	57140	15013
JNuke	20	2	4	4	335	117

prohibitively large in practice in all cases except for the last one. The complexity of DD is linear in the size of failed build/test runs [14]. However, in IDD, larger patches tend to be more vulnerable to patch conflicts in later revisions. Therefore, once a patch has accumulated a number of features such as refactorings, name changes, etc., the `patch` command tends to fail more often. As a result of this, the DD subroutine is used more often, which tends to lead to even larger patches.

Therefore, when patches get larger, both the size and the frequency of DD runs increases. This means that the number of test/build runs grows faster than linearly in the number of DD invocations. As each build/test cycle takes about 30 seconds in average, longer IDD runs can take several days.[4] In practice, the likelihood of a success is highest for quickly detected defects, so DD would typically be run for a few hours or days before it would be aborted.

7 Conclusion

Delta debugging automates the task of identifying minimal change sets. However, it requires a correct version, which may not be known when a new defect is discovered. Still, it is possible that an old revision passes a given test. Sometimes it is necessary to apply changes of newer versions to older ones in order to allow a newer test to execute on an old version. Iterative delta debugging automates this process and successively carries changes from newer versions back to older ones, until either a correct version is found or the process is aborted. The process is successful in some cases, but requires much time when change sets get large.

8 Future work

IDD works in conjunction with delta debugging, but requires a high-performance, high-precision implementation of DD in order to be successful.

The current tool chain has been written as a prototype, lacking several possible optimizations that could be implemented. For instance, DD generates and compiles variations of one revision by checking out a fresh copy of the necessary files and compiling

[4] Intelligent caching of compiled artifacts could reduce this time by a factor of 10. However, we did not find a system that supported all the programming languages and revision control systems in question.

them from scratch in each iteration. Whenever the build configuration does not change, most of these steps can be cached.

Furthermore, code analysis could eliminate the need to check variations where too much code is removed. Code coverage and dynamic slicing on a known "good" version can help to identify which statements are crucial for a test to pass. Such statements should not be removed in the DD process. For C programs, memory checking algorithms can further automate the suppression of programs that do not generate consistent results.

The direction of applying hierarchical DD is definitely promising, and delivers faster and better results than standard DD. However, the hierarchy of the patch file structure does not exactly mirror the hierarchy of software source code. Tools that represent software source code in XML format [2,6] could be used to extract a hierarchical representation of programming constructs. If XML data is used, then the question of how to generate an efficient and expedient difference representation is still open. Tools having their own format exist [4], and may be used in further case studies.

References

1. C. Artho, E. Shibayama, and S. Honiden. Iterative delta debugging. In *19th IFIP Int. Conf. on Testing of Communicating Systems (TESTCOM 2007)*, Tallinn, Estonia, 2007. Poster/Tools session.
2. K. Gondow, T. Suzuki, and H. Kawashima. Binary-level lightweight data integration to develop program understanding tools for embedded software in C. In *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 336–345, Washington, USA, 2004. IEEE Computer Society.
3. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
4. T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *Proc. 2006 ACM symposium on Document engineering (DocEng 2006)*, pages 75–84, New York, USA, 2006. ACM.
5. R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proc. 12th Int. Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 63–72, Newport Beach, USA, 2004. ACM.
6. K. Maruyama and S. Yamamoto. A tool platform using an XML representation of source code information. *IEICE – Trans. Inf. Syst.*, E89-D(7):2214–2222, 2006.
7. G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proc. 28th Int. Conf. on Software Engineering (ICSE 2006)*, pages 142–151, Shanghai, China, 2006. ACM Press.
8. G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
9. N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. 3rd Int. Workshop on Run-time Verification (RV 2003)*, volume 89 of *ENTCS*, pages 22–43, Boulder, USA, 2003. Elsevier.
10. D. Peled. *Software Reliability Methods*. Springer, 2001.
11. L. Torvalds and C. Hamano. git-bisect(1) manual page, 2008. `http://kernel.org/pub/software/scm/git/docs/git-bisect.html`.
12. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
13. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.
14. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering*, 28(2):183–200, 2002.