Making Automatic Repair for Large-scale Programs More Efficient Using Weak Recompilation

Yuhua Qi, Xiaoguang Mao* and Yan Lei
Department of Computer Science and Technology
National University of Defense Technology
Changsha, China
{*yuhua.qi, xgmao, yanlei*}*@nudt.edu.cn*

***Abstract*—For large-scale programs, automatically repairing a bug by modifying source code is often a time-consuming process due to plenty of time spent on recompiling and reinstalling the patched program. To suppress the above time cost and make the repair process more efficient, a recompilation technique called weak recompilation is described in this paper. In weak recompilation, a program is assumed to be constructed from a set of components, and for each candidate patch only the changed code fragment in term of one component is recompiled to a shared library; the behaviors of patched program are observed by executing the original program with an instrumentation tool which can wrap specified function. The advantage of weak recompilation is that redundant recompilation cost can be suppressed, and reinstallation cost will be cut down completely. We also built WAutoRepair, a system which enables scalability to fix bugs in large-scale C programs with high efficiency. The experiments confirm that our repair system significantly outperforms Genprog, a famous approach for automatic program repair. For the wireshark program containing over 2 millions lines of code, WAutoRepair spent only 0.222 seconds in recompiling one candidate patch and 8.035 seconds in totally repairing the bug, compared to Genprog separately taking about 20.484 and 75.493 seconds, on average.**

***Keywords*-weak recompilation; efficiency; automated program repair**

I. Introduction

Program repair is a tedious and difficult activity that requires lots of resources spent on locating and fixing program bugs [1]. And recently there is some promising work [2], [3], [4], [5], [6], [7] on automated program repair, reducing the cost of software maintenance. In general, the repair process can be divided into three phases: locate the program bug; generate the candidate patches in light of some specified rules; validate these patches through some test cases again and again until a valid patch is found. To the best of our knowledge, current works mainly focus on the second phase: how to generate the candidate patches. However, cost reduction techniques in the process of program repair are rarely considered. This paper seeks to reduce the computational cost by optimizing the program recompilation process.

*Corresponding author:
Email address: xgmao@nudt.edu.cn (Xiaoguang Mao)

A. Limitations of Existing Systems

Correcting a large-scale program with current approaches is often a time-consuming operation. In addition to the time spent in locating the defect code fragment, generating the candidate patches and running the patched program against test cases, a significant amount of time is spent in recompiling and reinstalling the patched program(see Figure 8 in [2]). When a candidate patch is generated, the target program with the application of the patch has to be recompiled and reinstalled to check whether the patch is valid or not [8]. But with so many patches to validate for a large-scale program, recompiling each of the patched programs leads to the high compilation cost, making the repair process less efficient. For example, time spent in fixing one bug in the **php**[1] is, on average, 1.84 hours [9]; recompilation of the patched **php** takes an average of 78.4% of the whole repair time [2]. As a result, for large-scale programs time for the program recompilation must be suppressed to make current approaches more scalable.

In recent approaches for automated program repair [9], [4], for each candidate patch both positive test cases characterizing the normal behaviors and negative test case exposing the fault are tested. The goal is to obtain a valid patch passing all test cases. Consider a candidate patch: first apply the patch to the source code of target program; utilize traditional compiler like GCC to recompile the source code to a new executable program; run against test cases to validate the patch. We note that with these approaches: all the files depending on the modified code fragment, even all the source files, in extreme cases, are recompiled; and sometimes the patched program has to be reinstalled and reconfigured in order to ensure that everything works well. This kind of recompilation technique described above will be referred as *strong recompilation*.

However, for strong recompilation redundancy is evident in most cases but becomes a much more serious problem for large-scale programs [10]. Since programmers are always

[1]Refer to the php interpreter implementation

978-1-4673-2312-3/12/$31.00

competent, the program developed by them are very close to the correct version [11]. Thus, it is reasonable to assume that most faults are merely simple ones, and can be fixed by some small syntactical changes. For automated program repair, a candidate patch often consists of only a few small changes to some code fragment [2]. Therefore, there is only minor code difference between each patched program and the original program, and recompiling each patched program completely in strong recompilation technique will suffer from redundant recompilation cost. For large-scale programs, suppressing the redundant cost, even to some extent, can make the repair approaches more efficient and scalable.

B. Contributions

This paper describes a recompilation method called *weak recompilation*, and presents a repair system with weak recompilation integrated. Suppose that P is the original program, that c is a component of P (i.e., the program P is assumed to be constructed from a set of components $C = \{c_1, c_2, \ldots, c_n\}$), and that there is a patch pt representing change transformation that can be applied to c to produce c'. Consider P' to be the patched P containing the component c' with the pt applied to c. In weak recompilation, Instead of recompiling all the source files desponding on the modified code fragment, we recompile only the changed component of c' to a shared library. Then, we use function wrapping, a mechanism provided by an instrumentation tool such as Valgrind [12], to implement the update to the original program. The implementation of weak recompilation depends on the specification of a set of components and the language in which the target programs are written. In general, components will correspond to elementary elements in target program with special language. Functions, classes and even files are all examples of components. Special to this paper, we have implemented weak recompilation on C language with our view of components: We define components as the functions included in source code.

The advantage of weak over strong recompilation is that weak recompilation requires less compilation cost. Specifically, benefits are twofold: first, the recompilation time with weak recompilation is suppressed and relatively stable. Since most programs are developed by competent programmers [11], then most bugs can be successfully fixed by modifying only some component of c in most cases. Thus in strong recompilation redundancy is obvious. However, Weak recompilation, which recompiles only the modified component, does not suffer from the redundancy problem no matter what the scale of the target program is. Second, in weak compilation the reinstallation time is cut down completely. Instead of reinstalling and reconfiguring the recompiled program in order to make the patched program work well in strong recompilation, we can make a patch validly work by loading the shared library produced in weak recompilation, and then implement the update by aid of function wrapping mechanism supported generally by instrumentation tools. As the original executable program is not regenerated or modified at all, the subsequent reinstallation and reconfiguration are not needed.

One point needs to be noted is that for weak recompilation the aid of the instrumentation tool is indispensable (hence, the term "weak"). The reason for that is its way of partial recompilation leads to that the original executable program is not modified at all. As a result, to make a patch work well the original program has to run with some instrumentation tool. But, the instrumentation tool will slow the execution of target program in some extent [13], making the execution of each test case more time-consuming. In this way, Weak recompilation sacrifices testing cost for cost reduction in compilation. That is, there is a tradeoff of recompilation cost versus execution cost. Generally speaking, for strong recompilation the compilation cost scales roughly linearly with the scale of program. In contrast, the execution cost running with instrumentation tool is not very sensitive to the scale of program. Thus the larger the scale of program, by and large, the better weak recompilation performs. We discuss the weak recompilation tradeoff and related empirical experiments in Section IV-B3.

With the above insights, in this paper we present a novel and scalable recompilation technique called weak recompilation, and implemented WAutoRepair, an automated repair system with the application of this technique. We also evaluated it on three real-world programs: **libtiff**, **php** and **wireshark** with real-life bugs taken from Claire Le Goues *et al.* [9]; compared to Genprog [2], a famous automated repair approach, our approach requires less recompilation cost, and thus has a higher repair efficiency.

In short, this paper makes the following contributions:

- We present the idea of weak recompilation for recompiling only the code fragment in term of one component (instead of the files depending on the changed code like strong recompilation used in current repair approaches) having been modified because of the application of the candidate patch, making the compilation cost in the repair process less and more stable regardless of the target program scale. In addition, in weak recompilation both reinstallation and reconfiguration costs are cut down, further reducing the computational cost of repair process.
- By combining the weak recompilation technique with Genprog, we have implemented WAutoRepair, a new system for automated C program repair. In the repair process, with the patched function-level component compiled using regular compilers (such as GCC) to a shared library representing the candidate patch, the original executable program is not modified or regenerated at all.
- Experimental results demonstrates that WAutoRepair is apparently time-saving for fixing the bugs existing in

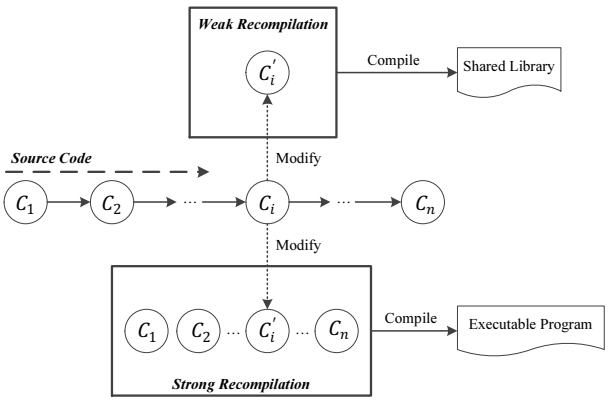

Figure 1. Weak recompilation versus strong recompilation

large-scale programs, compared with Genprog. Furthermore, for WAutoRepair the compilation cost is relatively stable regardless of the scale of target programs, making our system more scalable and practical.

II. Weak Recompilation

In this section we describe weak recompilation in detail. We show the difference between weak recompilation and strong recompilation (Section II-A), discuss the types of program components (Section II-B), and present the patch representation (Section II-C).

A. Overview

Weak recompilation supports flexible program recompilation that allows only the changed code fragment to be recompiled. As described in Figure 1, suppose that the source code of a program P is constructed from a set of components $C = \{c_1, c_2, \ldots, c_i, \ldots, c_n\}$, and with existing fault localization technique [14] the component of c_i is located as the defective one causing a failure . Then, c_i is modified to repair the program, and thus corresponding c_i' is obtained. In weak recompilation, instead of recompiling all the relevant components to a new executable program like strong recompilation used by overwhelming majority of current repair approaches, the c_i' component needs only to be compiled to a shared library. Clearly, weak recompilation requires less program compilation cost, and thus is more time-saving than strong recompilation.

B. Program Components

In this paper, the C component preliminarily refers to one of three types: function, class and file. The definition of program component depends on both the language and the accuracy of the corresponding fault localization used to locate the cause of a failure. For example, for the C programs, the type of class does not need to be considered. For the C++ programs, because in weak recompilation only one component is allowed to be recompiled, with the file-level component it is more likely to successfully fix bugs even with relatively bad fault localization accuracy (meaning more suspicious statements which are irrelevant to bugs), although the function-level or class-level weak recompilation requires less compilation cost.

In general, the function-level weak recompilation should be suitable in most cases. Many bugs are merely a few simply program faults which can be fixed by a few trivial changes, and these changes often locate in the same function, which may be tracked down with relatively high accuracy through existing fault localization techniques. We found that all the 55 bugs can be fixed within one function after we reproduced all these experiments on Genprog [9]. That is, at least half of existing bugs (55 of 105) can be fixed within one function, since the benchmark programs and defects are indicative of "real-world usage" [9]. Hence, to some extent it is reasonable to limit the the modifiable code scope in one function body, and is probable to successfully repair the defective program with higher efficiency in function-level weak recompilation.

C. Patch

As described in Figure 1, we try to modify the suspicious component of c_i in order to repair a defective program P. In reality, in the process of repair, each c_i' component corresponding to the modified c_i serves as one concrete patch in term of shared library. This section mainly describes the structure, automatic generation and the application of patches in weak recompilation.

1) Patch Description: Instead of merely describing the literal changes in the source code like traditional patch [5], a patch produced by weak recompilation includes full code of the c_i' component, which will subsequently take the c_i's place to be called by the original program to check whether the patch is valid or not. In addition, to ensure that the patch can be correctly compiled to shared library without errors, a patch also needs to include some necessary context information such as type definition and variable declaration.

2) Patch Generation: To ensure that the whole repair process is automatic, we have implemented a patch generator, which is based on CIL-1.3.7 in OCaml language and has been integrated into the WAutoRepair system, to construct patches for C programs in function-level weak recompilation. Taking as inputs the component (function) name, according to specified rules this generator can automatically produce the patch file by extracting relevant information from the corresponding source code. The process of patch generation is presented in Figure 2.

First, Generate the intermediate files from source code. For most large-scale programs, the recompilation of one function often depends on some context information included in other files. For simplicity, for each file included in source code we extract and write both itself and relevant context information to a new file called intermediate file (*.i* file in Linux) using GCC with the flag *--save-temps*. In fact,

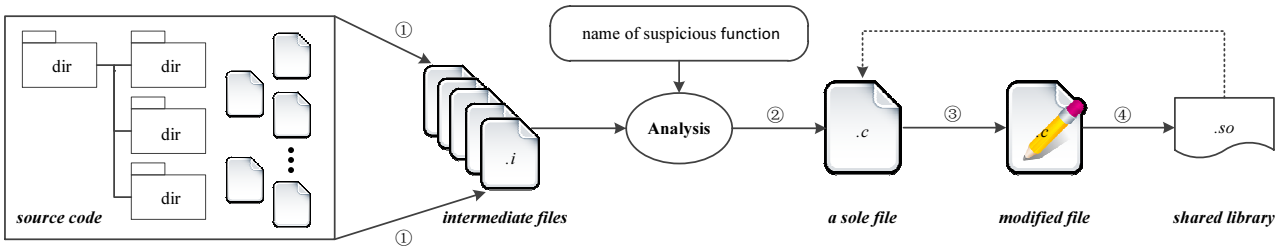

Figure 2. The process of generating a patch

each of intermediate files is the same as the corresponding file in original program but the extra context information.

Second, produce a sole file constructed from both the suspicious function and context information, which subsequently assists the function to be successfully compiled to a shared library, independently. Taking the function name and intermediate file including the definition of this function, our generator can produce the file of full automation.

Third, modify the code of the function body in the file generated in the second step. Following the insight described in Section II-A, in light of some modification rules we seek to repair the defective program by modifying merely the suspicious component in term of function.

At last, compile using GCC the changed file to a dynamic patch file in term of a share library. Note that the last two steps can be repeated or iterated to produce more candidate patches.

3) Patch Application: Utilizing function wrapping mechanism generally provided by instrumentation tools, we have realized the updating of dynamic patch files. When a function is wrapped by aid of instrumentation tools, then it means that calls to the function are intercepted and rerouted to another specific function. Most instrumentation tools such as Valgrind can monitor and control the execution of target program. With the original program executed with instrumentation tools, we can wrap suspicious function, so the calls to the suspicious function are intercepted and rerouted to the corresponding function included in the shared library produced in Section II-C2. From the viewpoint of the user, it is more likely that the patched program rather than original one is executed. By this way, we deliver candidate patches, in the forms of shared libraries, to real, large-scale and complex programs without recompiling it completely.

III. WAutoRepair

For maximum applicability we implemented a new repair system called WAutoRepair, which enables scalability to fix bugs in large-scale C programs with high efficiency, and the framework of WAutoRepair is presented in Figure 3. In this section, we describe Genprog on which WAutoRepair is built, present the framework of WAutoRepair in detail, and then provide a case study that uses WAutoRepair to fix one real bug in the **php**.

A. Genprog

Genprog[2], a generic method for automatic software repair, can repair defects in deployed, legacy C programs without formal specifications [2], [9]. With the hypothesis that some important functionalities missed by a program will be able to occur in another position in the program, Genprog can repair program by copying a few lines of code from other parts of the program or modifying existing code. Genprog takes advantage of a modified version of generic programming to maintain a population of variants which are represented by the corresponding pairs of abstract syntax tree (AST) and weighted path. Specifically, with a stochastic process, a candidate patch is generated using one of generic algorithm operations: mutation and crossover; the variant is just the AST representing program with the patch applied. Then, to validate this patch, test cases are required: firstly, the variant is compiled, and fitness is evaluated by executing test cases. If some patch passes all the test cases, Genprog terminates the process of searching and declares that a valid patch is found.

As an innovative method, Genprog has successfully fixed many bugs existing among some off-the-shelf programs (e.g. php, libtiff). However, Genprog suffers from the scalability problem, that is, for large-scale programs, time for recompiling a patched program will increase rapidly, making the repair process less efficiency. And we try to address this problem by applying weak recompilation.

B. Framework

We built WAutoRepair in OCaml language based on Genprog. In the framework of WAutoRepair, with *function-level* weak recompilation integrated, WAutoRepair fixes a bug the way like that: when a failure occurs, the fault information is obtained through the fault localization component; with the fault information in term of the name of suspicious function, the patch generation component automatically generates a shared file (i.e., the patch file) according to the modification rules used by Genprog; with the shared library LD_PRELOADed, we can get the behavior of patched program by executing the original program with Valgrind, and then check whether the patched program works well by

[2]Available: http://church.cs.virginia.edu/genprog/index.php/Main_Page

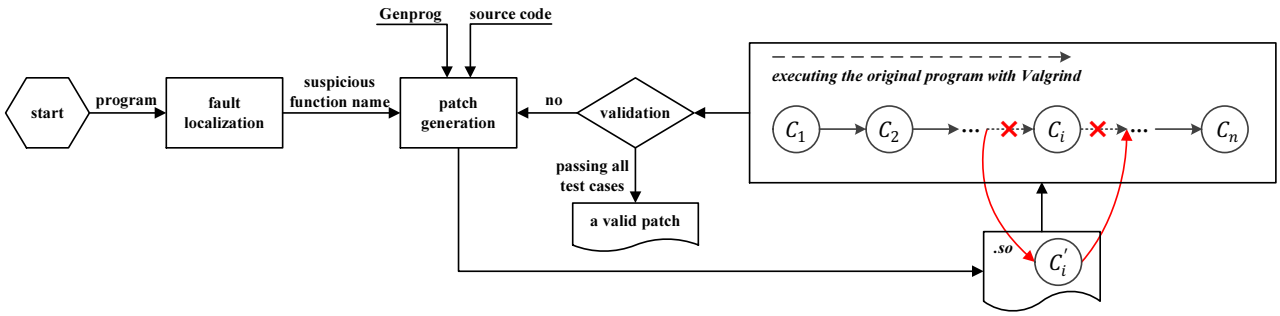

Figure 3. The framework of WAutoRepair

running against test cases. If the patched program passes all the test cases, then a valid patch is found, otherwise WAutoRepair has to go back to the component of patch generation and continue. WAutoRepair repeats the search process until a valid patch is found, or when limitations are arrived (namely, too much time or too many tries elapse). More details for these components in Figure 3 are illustrated in the following subsections.

1) Fault Localization: In our framework described in Figure 3, we do not assign the specific approach for the component of fault localization, although a lot of noteworthy works have been done for fault location [15], [16]. We assume that the defective areas have been already located. What we need to do is just to find the function in which the defective areas locate. This function is called *suspicious function*, and transferred to the component of patch generation for generating candidate patch through the modification of suspicious function. Therefore, the accuracy of fault localization plays a very important role in the effectiveness and efficiency of our WAutoRepair. In the paper, however, fault localization is not the focal point, since so many works have been done for that.

2) Patch Generation with Genprog: Recall that we modify the suspicious function code according to the modification rules (see Section II-C2). Special to this framework, we take the rules used by Genprog to guide the modification process. Note that the rules which can be utilized by WAutoRepair are not limited to Genprog, WAutoRepair also works well with other rules [4], [5].

In consideration of the scalability problem, our framework merely borrows above modification rules to guide the way to generate patches. As long as a patch is generated, a new way described in following section, instead of Genprog's method, will be explored to validate the patch.

3) Executing the Program with Valgrind: As the patched program is not recompiled completely, we obtain the execution behaviors of patched program by executing the original program with an instrumentation tool, which provides the mechanism to wrap a specific function.

WAutoRepair uses Valgrind[3] as its instrumentation tool. Valgrind can instrument and monitor both C and C++ programs. In addition, compared to other instrumentation tools, wrapping specifications and semantics for Valgrind to wrap functions are relatively simpler. This is also one important reason for our selection of Valgrind.

Valgrind can automatically complete most works for function wrapping. What we need is to generate the wrapper which is represented by a function with a special name which identifies itself as the wrapper for the wrapped function; then compile the library of wrappers to object code in the normal way. With no need for an external script, Valgrind distinguishes by itself which wrapper corresponds to which original function. In fact, the patched function in the shared library (i.e., *.so* file in Figure 3) is just the wrapper written with the wrapping mechanism; then by executing the original program with Valgrind the behaviors of program with the patched function are observed.

4) Validation: Like relevant work [17], [9], [4], there is no priori guarantee that the generated patches do fix bugs. Hence, when a candidate patch is produced by the component of patch generation, we need to check whether the patch is valid or not. Consider a patch file in term of a shared library, which is produced by the component of patch generation. The validation component runs the patched program (see Section III-B3) against both the set of positive test cases T_P characterizing the required behaviors of the program and the set of negative test cases T_N encoding the fault to be repaired. WAutoRepair considers the patch to be valid only if the patched program successfully passes all the test cases in $T_P \cup T_N$, otherwise WAutoRepair go back to the component of patch generation and continue the search process until some limit is arrived.

C. Case Study

The **php** program, an excellent interpreter for a web-application scripting language, is popular and widely-used. However, there is a bug existing in the version 5.2.1: when the *str_replace* function is called in such a way like this [2]:

[3]Available: http://valgrind.org/downloads/current.html

```
 …
void php_str_replace_in_subject(zval *search, zval *replace, …
 {
    …
  if (Z_STRLEN_P(search) == 1) {
     php_char_to_str_ex(
     Z_STRVAL_PP(subject),  Z_STRLEN_PP(subject),
     Z_STRVAL_P(search)[0], Z_STRVAL_P(replace),
     Z_STRLEN_P(replace), result, case_sensitivity, replace_count);
  } else if (Z_STRLEN_P(search) > 1) {
     Z_STRVAL_P(result) =
      php_str_to_str_ex(Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
      Z_STRVAL_P(search), Z_STRLEN_P(search),
      Z_STRVAL_P(replace), Z_STRLEN_P(replace),
      &Z_STRLEN_P(result), case_sensitivity, replace_count);
      } else {
              *result = **subject;
              zval_copy_ctor(result);
              NIT_PZVAL(result);
           }
 …
```

(a) The *php_str_replace_in_subject* code fragment in php

```
1 #include "valgrind.h"
2 #include </home/gracegenprog_php/test_php/string.c>

3 int I_WRAP_SONAME_FNNAME_ZU(NONE,php_str_replace_in_subject) (zval
  *search,  zval *replace, zval **subject, zval *result, int case_sensitivity, int *replace_count)
4  {
5    …
     if (Z_STRLEN_P(search) == 1) {
        php_char_to_str_ex(
        Z_STRVAL_PP(subject),  Z_STRLEN_PP(subject),
        Z_STRVAL_P(search)[0], Z_STRVAL_P(replace),
        Z_STRLEN_P(replace), result, case_sensitivity, replace_count);
     } else if (Z_STRLEN_P(search) > 1) {
        Z_STRVAL_P(result) =
         php_str_to_str_ex(Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
         Z_STRVAL_P(search), Z_STRLEN_P(search),
         Z_STRVAL_P(replace), Z_STRLEN_P(replace),
         &Z_STRLEN_P(result), case_sensitivity, replace_count);
         } else {
                 *result = **subject;
                // zval_copy_ctor(result);
                // NIT_PZVAL(result);
              }
     …
     }
```

(b) The corresponding *wrapper.c* file

Figure 4. The php case

str_replace("A", str_repeat("B", 65535),
str_repeat("A", 65538));

Php will be abnormally aborted with an integer overflow fault.

To fix the bug, WAutoRepair first needs to obtain information for the defective code fragment causing the fault. By aid of typical tools of fault localization such as Tarantula [18] or even debugging experiences, the function of *php_str_replace_in_subject* probably contains a bug and so is considered to be the suspicious function. Figure 4a provides the **php** program fragment with the *php_str_replace_in_subject* function.

Taking the name of suspicious function and source files, the component of patch generation first analyses source code and produces a intermediate file, which is constructed from both the *php_str_replace_in_subject* function and relevant context information; then generates one patch at a time by modifying the body code of *php_str_replace_in_subject* according to the modification rules provided by Genprog. One example patch is shown in Figure 4b. Now, we describe the details of the patch.

As shown in Figure 4b, the candidate patch is mainly made up of two parts: one is the macro with the name of *I_WRAP_SONAME_FNNAME_ZU* (line 3-4) from the included *valgrind.h* (line 1). It is an encoded name which Valgrind notices when reading symbol table information. By this way, the calls to *php_str_replace_in_subject* are automatically intercepted and rerouted to the function encoded by the macro in the loaded library if the program is executed with Valgrind; the other one is the function body (start from line 5) really executed in the subsequent validation process. Specific to this example, the function body is just the original body with the candidate patch applied. In addition, to ensure that the file can be compiled to a dynamic link library successfully, line 2 includes some relevant context information required by the function body.

Once one candidate patch is produced, WAutoRepair immediately compiles it to a shared library with the file name *wrapper.so*. Then with *wrapper.so* loaded, WAutoRepair runs the original **php** program with Valgrind against test cases in the validation process. By this way, the candidate patch in term of *wrapper.so* is applied to the **php** program. For the purposes of this example, one negative test case reproducing the fault and three positive test cases encoding the required behaviors are used to validate the candidate patch. Note that we can increase the number of test cases to guarantee that a patch is actually valid with high confidence level. If the patch passes the negative test case and still passes all the positive test cases, WAutoRepair considers the patch is valid and terminates the repair process, otherwise WAutoRepair goes back to the component of patch generation and continue the search process until some limits are arrived.

Ultimately, WAutoRepair successfully fixes the bug and requires 14.243 seconds on average. The detailed performance study on WAutoRepair is described in the following section.

IV. Experimental Evaluation

This section systematically evaluates and analyzes the experimental results of using WAutoRepair to repair three representative programs with the scales (expressed by the lines of the source code) differing by orders of magnitude. To demonstrate the advantages of WAutoRepair, we also show the experimental results of Genprog with the same experimental context.

Table I
EXPERIMENTAL RESULTS

Program	LOC	Approach	Validating ONE Candidate Patch				Others(s)	Total(s)
			Compilation(s)	*Positive test cases(s)*	*Negative test case(s)*	*Sum(s)*		
libtiff	84,077	*WAutoRepair*	*0.128*	*0.800*	*0.675*	*1.603*	*0.545*	*6.951*
		Genprog	12.315	0.032	0.109	12.456	0.447	13.152
php	764,489	*WAutoRepair*	*0.760*	*2.373*	*0.806*	*3.939*	*3.488*	*14.243*
		Genprog	17.391	0.124	0.049	17.564	5.317	69.600
wireshark	2,814,000	*WAutoRepair*	*0.222*	*2.131*	*0.559*	*2.912*	*1.600*	*8.035*
		Genprog	20.484	0.830	5.651	29.965	1.879	75.493

A. Experimental Setup

We selected the **libtiff**, **php** and **wireshark** programs with three real bugs[4], taken from Claire Le Goues *et al.* [9], as the experimental benchmarks. All the three programs are written in C language, and widely-used in the world. For each bug, we selected one negative test case and three positive test cases, and tried to run WAutoRepair to repair them the way similar to aforementioned case in Section III-C. WAutoRepair will terminate itself if a valid patch is found, or when some limits are arrived (over 100 trials or 3 hours elapse).

For the purposes of comparison, all the experimental parameters for Genprog (including both the Genprog rules used by WAutoRepair and the original Genprog) in our experiment are the same as those settings in [2] all but the weighted path, a sequence of <statement, weight> pairs. In our experiment the weighted path, in which the default weight 1.0 is assigned to each element, is exactly the set of all the statements included in the suspicious function. Namely, the genetic operations of mutation and crossover are constrained to operate only on the suspicious function, and that is consistent with the assumption of weak recompilation.

All the experiments ran on an Ubuntu 10.04 machine with 2.33 GHz Intel quad-core CPU and 4 GB of memory. For each program, we performed 100 trials, and only recorded the trials leading to a successful repair.

B. Results

All the experimental results are summarized in Table I. The repair effectiveness of WAutoRepair is almost the same as Genprog, because WAutoRepair seeks to search the candidate patches according to the modification rules used by Genprog. And detailed discussion for effectiveness can be obtained in [19]. In this section we mainly analyze and compare the performance between WAutoRepair and Genprog aiming to address the following three questions:

- How much the performance overhead can be reduced by WAutoRepair compared to Genprog?
- What determines the effectiveness of weak recompilation?
- What and how much is the performance loss incurred by the instrumentation tool?

1) Performance Analysis: In this paper, we preliminarily measure the performance overhead in term of time spent in repairing each bug. As shown in Table I, the "LOC" column gives the scale of each program in the form of lines of source code; the "Validating One Candidate Patch" column reports the average time overhead for validating ONE candidate patch, in terms of "Compilation" (the average time spent in compilation for each successful trial), "Positive test cases" and "Negative test case" (the average time taken for the execution of test cases in a successful trial), "Sum" (the whole time overhead to validate one patch for each repair); the "Others" column gives the average time overhead spent on the other parts such as the modification to generate each successful patch. At last, the "Total" column reports the complete time overhead carried to find a valid patch.

Clearly, for the time overhead as a whole, with much smaller values in the "Total" column WAutoRepair performs obviously better than Genprog, especially for the **wireshark** program containing over two millions lines of code: the time for running WAutoRepair to repair the bug existing in **wireshark** is only *11%* of that for running Geonprog; Genprog further spent over 92 times of time on the compilation of one candidate patch than WAutoRepair, on average. In addition, with the application of weak recompilation WAutoRepair performs more steady with less compilation cost regardless of the scales of target programs. For example, even with the scales differing by orders of magnitude, for all the three programs the time overhead spent in compilation of one candidate patch is less than 1 seconds. However, due to the performance overhead incurred by function wrapping mechanism, WAutoRepair requires extra performance overhead in the form of larger time overhead for the execution of test cases. Fortunately, for large-scale programs, this extra performance overhead is often relatively minor in contrast to the reduced compilation overhead. We defer discussing this issue to Section IV-B3.

2) Effectiveness of Weak Recompilation: As described in Table I, our weak recompilation is very effective in

[4]Available: https://church.cs.virginia.edu/genprog/archive/genprog-105-bugs-tarballs/

reducing the compilation overhead, and relatively stable no matter what the scale of the target program is. Then, what determines the effectiveness of weak recompilation? More specifically, what is the condition under which WAutoRepair has the greatest advantage over Genprog?

Intuitively, for traditional strong recompilation recompilation overhead is highly correlated with both the scale of target program and the number of files depending on the defective code. The larger the scale and the number, the more recompilation overhead is likely performed. However, for weak recompilation the recompilation overhead is weakly correlated with the above two factors, because only minor code fragment (in term of function-level component) is recompiled, in fact. As a result, it is reasonable to assume that: the larger the program scale or the more files depending on the defective code, the more effectiveness of weak recompilation.

3) Instrumentation Overhead: As discussed previously, as WAutoRepair implements the function wrapping mechanism by aid of Valgrind (a instrumentation tool running with the target program), WAutoRepair carries the extra instrumentation overhead incurred by Valgrind. Concretely, the function wrapping mechanism makes the target program execute more slowly, leading to a relatively large execution cost. As shown in Table I, for WAutoRepair all the execution cost (both "Positive test cases" and "Negative test cases") have larger value than that of Genprog except the negative test case in **wireshark**. The exception is caused by the mechanism of memory leak detection through the analysis to the repair process of **wireshark**.

There is a clear tradeoff of recompilation cost versus execution cost, and Figure 5 shows the results by comparing the overhead between WAutoRepair and Genprog. Because the execution cost of negative test cases is susceptible to bugs, so we selected the execution cost of positive test cases instead of the whole execution cost as the reference object for compare in Figure 5. For all the three programs, WAutoRepair has a obvious advantage in high compilation efficiency up to 96.2 times faster than Genprog; instrumentation overhead can incur no more than 25 times over performance overhead relative to the original execution cost of test cases; and furthermore, instrumentation overhead ratio does not scale up with the rising code lines of the target program. For example, for **wireshark** with the largest scale of the three programs, running WAutoRepair to repair the bug suffers from approximate 2.6 times more execution cost than running Genprog, but for **libtiff** with the smallest scale, the corresponding value is 25 times! Given that a significant amount of time overhead is spent on recompilation for large-scale program (for **wireshark** the ratio is 74%), it is reasonable to assume that WAutoRepair is more likely effective in improving the repair efficiency for large-scale programs.

In addition, we can increase the number of test cases to guarantee that the obtained patch is actually valid with high confidence level. In this case, WAutoRepair can alleviate the extra execution overhead with the application of sampling technique [9], which first evaluates the patched program on a random sample of test cases before the evaluation on the total test cases.

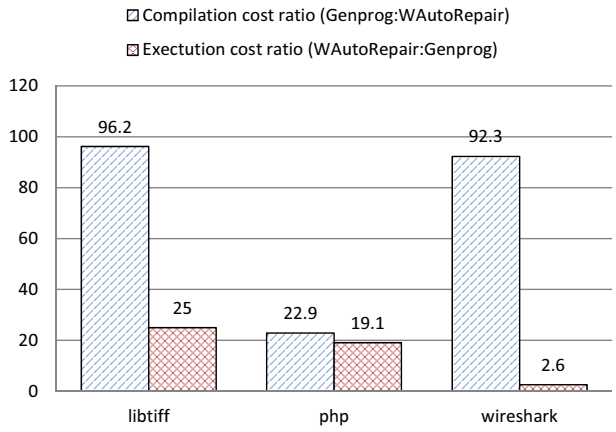

Figure 5. Comparing the overhead

C. Threats to Validity

Since only 3 programs are investigated, conclusions drawn from the experimental results are weak and hardly generalizable. But we consider that our evaluation is believable and reliable for at least three points: the bugs existing in the post-release programs are real-life; the selected benchmark programs are good representations of large-scale programs differing by orders of magnitude; at last, Genprog compared with WAutoRepair in the experiment is one of the most recent and noteworthy work for automated program repair [20]. We plan to strengthen the conclusions through more experiments on many programs in the future work. In addition, if the test cases selected to validate the candidate patch is long-running, the advantage of WAutoRepair may be not very evident. Fortunately, we can use the checkpoint tool [21] to suppress, at least partly, the long-running execution cost.

V. Related Work

A. Automatic Program Repair

Because the candidate patches are generated in light of modification rules used by Genprog in WAutoRepair, the recent work by Werimer *et al.* [9], [2], [19], [8], [22] is most related to this paper. They proposed the Genprog, an automated approach for fixing bugs existing in legacy programs, and utilized an extended form of genetic programming to get a program variant. Arcuri [17] firstly proposed the idea of fixing software with search algorithms, and shown some preliminary experimental results on automatically fixing a defective sorting routine. However, the experimental results

in his work were not reported on real-world software with real bugs. Hence, the effect of their works in practical application is still unknown. Another noteworthy work on automatic program repair is done by Yi Wei *et al.* [23], [4]. They present a novel technique for automatically fixing defective programs with contracts [24].

Currently, the three approaches are the popular works for automatic program repair at the source code level. Although they generate the candidate patches according to different rules, there is at least one common point: to validate a patch, they have to execute the patched program through all the test cases. And all these approaches scale not very well with the large-scale programs. To address the scalability problem, similar to the approach presented in this paper, we can also apply weak recompilation technique to these approaches to suppress time spent on validation.

B. Cost Reduction Techniques for Mutation Testing

Similar to automated repair, mutation testing also needs to recompile the modified source code (a detailed account can be found in the survey [25]). As a relatively mature testing technique for evaluating the effectiveness of a test set in terms of its ability to detect faults, there are lots of techniques for cost reduction research work such as weak mutation and Compiler-Integrated technique.

Weak Mutation: Similar to weak recompilation technique, in weak mutation [26] a program P is also divided into lots of components, each of which referred to one of these types: variable, arithmetic expression and relation, and boolean expression. Consider that the c_i component is changed for mutation testing. Instead of executing the whole program and then checking the output like strong mutation, in weak mutation the program is terminated immediately after the execution of the c_i component. And the mutant is killed by comparing the program state of the mutant program with the original program state on the same test case. Clearly, weak mutation reduces the execution cost and improves the test effort but with less test effectiveness. The reason for that is the final output of the mutant may still be right even with different intermediate states. Hence, a natural next step would be to reduce the execution cost for repairing program in light of the insight of this method. However, there is a risk that some valid patches may be taken for invalid ones, leading to false negative.

Compiler-Integrated Technique: In mutation testing, the mutant is almost the same as the original program but with merely a minor difference, in most cases. To suppress the redundant compilation cost DeMillo *et al.* introduced an method to integrate support for program mutation directly into a compiler [27]. And this Compiler-Integrated Technique works like that: taking as inputs the original program, the compiler produces two outputs: a target executable program and a collection of additional machine instruction sequences. And each patch, in terms of one or more of these instructions, can be applied to the target executable program to yield a patched program without recompilation. Clearly, this technique can improve test efficiency by reducing the compilation cost. However, with the technique the compiler has to be modified, and that may lead to the incompatibility problem. In this paper, with no need for the modification to compiler, we suppress the redundant compilation cost with our weak recompilation technique.

C. Incremental Compilation

Currently, there is significant work in incremental compilation in areas like Integrated Development Environments where the data structures used by the compiler are persistent. As the original program is changed, though only the parts of the executable are changed, the subsequent reinstall cost, in some cases, is still necessary, because some executable programs are not invisible unless they are explicitly installed.

To the best of our knowledge there is little research for cost reduction technique on automatic program repair, although lots of studies for automatically finding and repairing program bugs. The body of this work focuses on the problem of compilation cost reduction in the repair process, which becomes much more important for large-scale programs.

VI. Conclusion

For the bugs exsiting in large-scale programs, current approaches for automated program repair often suffer from the high computational cost, a significant amount of which arises from the expensive compilation cost. To address this problem, we present the weak recompilation technique. Instead of recompiling all the relevant files like popular approaches, in weak recompilation we recompile merely the changed component of the target program. We also built a system called WAutoRepair with the application of weak recompilation technique. And the experimental results on three large-scale programs, one of which containing over two millions lines of code, reveal that WAutoRepair is very effective in reducing the redundant compilation cost, and gracefully outperforms the original Genprog. Complete experimental results in this paper are available at:

http://sourceforge.net/projects/wautorepair/files/

To our knowledge, for automated program repair we are the first to present and try to address the problem of expensive recompilation cost for large-scale programs.

Due to relatively high efficiency by applying weak recompilation, WAutoRepair spends less time on validating a candidate patch. That is, more candidate patches can be validated within a limited time bound; and thus it is possible that more complex modification rules (meaning that more trials) can be adopted to repair defective programs with large scales by WAutoRepair. This is also our future work.

ACKNOWLEDGMENT

The authors thank W.Weimer *et al.* for their noteworthy work on Genprog, based on which we built the WAutoRepair system. This work was supported by the National Natural Science Foundation of China under Grant No.90818024, 91118007 and 61133001, the National High Technology Research and Development Program of China (863 program) under Grant No.2011AA010106 and Program for New Century Excellent Talents in University.

REFERENCES

[1] R. C. seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54 –72, 2012.

[3] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494 – 3514, 2011.

[4] Y. Pei, Y. Wei, C. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *International Conference on Automated Software Engineering*, 2011, pp. 392 –395.

[5] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2011, pp. 1427–1434.

[6] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *International Conference on Software Testing, Verification and Validation*, 2010, pp. 65 –74.

[7] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Programming Language Design and Implementation*. New York, NY, USA: ACM, 2011, pp. 389–400.

[8] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *International Conference on Software Engineering, (to appear)*, 2012.

[10] C. Byoungju and A. P. Mathur, "High-performance mutation testing," *Journal of Systems and Software*, vol. 20, no. 2, pp. 135 – 152, 1993.

[11] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34 –41, 1978.

[12] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Programming Language Design and Implementation*. New York, NY, USA: ACM, 2007, pp. 89–100.

[13] N. Nethercote, "Dynamic binary analysis and instrumentation," Ph.D. dissertation, University of Cambridge, 2004.

[14] Z. Fry and W. Weimer, "A human study of fault localization accuracy," in *International Conference on Software Maintenance*, 2010, pp. 1 –10.

[15] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 76–87.

[16] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 121–130.

[17] A. Arcuri, "On the automation of fixing software bugs," in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 1003–1006.

[18] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 273–282.

[19] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374.

[20] M. Harman, "Automated patching techniques: the fix is in: technical perspective," *Communications of the ACM*, vol. 53, no. 5, pp. 108–108, 2010.

[21] O. Laadan and S. E. Hallyn, "Linux-cr: Transparent application checkpoint-restart in linux," in *The Linux Symposium*, 2010.

[22] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2010, pp. 965–972.

[23] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2010, pp. 61–72.

[24] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 191–200.

[25] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649 –678, 2011.

[26] W. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371 – 379, 1982.

[27] J. r. Edward William Krauser, "Compiler-integrated software testing," Ph.D. dissertation, Purdue University, 1991.