JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 0000; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Pragmatic Source Code Reuse via Execution Record and Replay

Ameer Armaly *, Collin McMillan

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 46556, US

SUMMARY

A key problem during copy-paste source code reuse is that, to reuse even a small section of code from a program as opposed to an API, a programmer must include a huge amount of additional source code from elsewhere in the same program. This additional code is notoriously large and complex, and portions can only be identified at runtime. In this paper, we propose execution record/replay as a solution to this problem. We describe a novel reuse technique that allows programmers to reuse functions from a C or C++ program, by recording the execution of the program and selectively modifying how its functions are replayed. We have implemented our technique and evaluated it in an empirical study in which eight programmers used our tool to complete four tasks over four hours each. The participants found our technique to be easier than manually reusing the code as part of their project. We also found that the resulting code was smaller and less complex than it would have been had the participants manually reused the code. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

1. INTRODUCTION

Source code reuse has long been a centerpiece of software engineering research [61]. This research has yielded many effective solutions for designing reusable software, such as object-oriented architectures [44] and repositories of shared libraries [46]. While these technologies have proliferated, the reality is that much source code is not designed with reuse in mind [40]. As a result, *pragmatic* reuse (also called "copy-paste" or "opportunistic" reuse [52]) has become an accepted practice in many professional environments [48, 45]. Pragmatic reuse differs from library or component reuse in that there is no interface to the reused code – the code must be transplanted from one program into another.

A key problem during this transplanting process is that the reused code's dependencies must also be transplanted. As Section 2 will show, to reuse even a small section of source code, a programmer often needs to include a huge amount of source code dependencies from elsewhere in the same project. The complexity and size of these dependencies (typically totaling 30% to 60% of the original program [8]) forces programmers to choose: either cut the reused code to reduce dependencies, or abandon the reuse altogether [40]. This difficulty of understanding dependencies is a consistent theme in studies of software reuse [83, 19].

Another problem faced during the transplanting process is that some code requires very specific and/or older versions of the compiler or other external dependencies. This requires that the programmer understand the exact nature of the differences between the versions required by the code that he or she wishes to transplant and the versions currently available. This can mean understanding the inner workings of the external dependency, e.g. when code relies on undocumented and

*Correspondence to: Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 46556, US E-mail:aarmaly@nd.edu

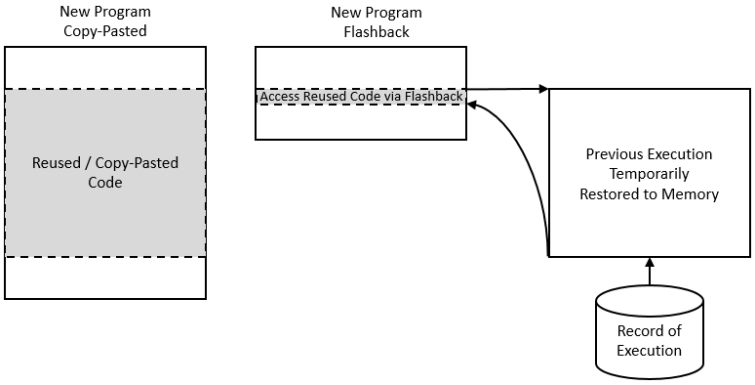


Figure 1. High-level overview of our approach. Instead of copy-pasting code into a new program, `Flashback` makes it possible to reuse code from previously-recorded executions. The two main advantages are: 1) the new program can be made much smaller, and 2) there is no need to port legacy code during reuse. Section 2 provides and in-depth example.

unsupported behavior that has changed. The programmer must then find a way to compensate for the changes in the external dependency in order to transplant the code. While migrating the code the programmer may encounter a dependence cluster [43] which is a set of statements that depend on each other. Changing any one of these statements can affect the other statements in the cluster, forcing the programmer to contend with modifying more statements in the cluster until every statement in the cluster is migrated to the new environment.

In this paper, we propose using *execution record/replay* to reduce the number of dependencies that a programmer must consider when reusing source code. Execution record/replay is the task of logging the execution of a program, so that the execution can be duplicated later [56, 80]. Our technique uses this idea by recording the state of the dependencies during one program's execution, and replaying them in the context of a different program. The advantage is that a programmer only needs to include the code that he or she would like to reuse. If that code has dependencies, then those dependencies are restored from the execution log as they are needed. The programmer can elect to modify these dependencies if desired, otherwise the functionality is available as it was in the original program. Figure 1 depicts our idea.

We have implemented our technique as a library called `Flashback` for reusing functions from C/C++ programs in a Linux environment. Using `Flashback`, a programmer can select a function in a program that he or she would like to reuse. Then, the programmer executes the program. `Flashback` records the state of the program into a "scene" any time that control is passed to the function. In cases where the function is called multiple times there will be multiple scenes for that function. The programmer will typically change any state information prior to execution. In cases where state information is important the programmer will need to keep track of which invocation contains the appropriate state. To reuse the function, the programmer directs `Flashback` to reload that state at a given point in the new program by calling the `flashback_load_scene()` function from within the host program. The programmer can then alter the state as necessary, such as the values of the function's arguments. At runtime, `Flashback` makes these changes, restores the state, and passes control to the function. Once the function is complete, `Flashback` passes control back to the new program. The new program does not need to include any of the reused function's dependencies; instead, they are restored from the scene.

To evaluate our technique, we performed an empirical study in which eight programmers used our technique to solve four programming tasks. Each programmer spent approximately four hours solving these tasks. We then compared the time required, size of resulting programs, and programmer-reported difficulty when using our technique versus a baseline copy-paste technique, which is the typical strategy followed by programmers (see Section 7). We found that our approach reduced the time required to solve the tasks by a statistically-significant margin. In addition, the resulting programs were smaller by a statistically-significant margin, and the perceived difficulty was reduced.

DOI: 10.1002/smr

2. BACKGROUND

This section provides an example demonstrating the problem and a description of execution record/replay technology.

2.1. Motivating Example

Consider the following reuse scenario from the open-source program Celestia[†], which we use as an example to explain our approach in Section 3. Celestia is planetarium software that shows a graphical display of the night sky. Consider a programmer who wants to use code from this application to calculate the star nearest to another, arbitrary star that the programmer specifies. The programmer believes this is possible because the application's graphical user interface contains a `Find Stars` window which performs this task. There are a number of successful techniques to help the programmer find relevant code for this task, and it is likely that the programmer would find the `nearestStar` function, which implements the search algorithm behind the `Find Stars` window. To reuse this function, shown in Figure 2, the programmer needs to understand several details behind Celestia. The programmer must know how it represents space, including the `Universe` and `Star` data types. The programmer must investigate the `Predicate` structure, the position variable `pos`, and even the parameters to the function `findStars`. To include `nearestStar` in another program, all of this additional source code would need to be extracted and included in the new program. Then initialization functions, such as the one that sets up the star catalog, also need to be added, along with any external files which the initialization functions use. The programmer would need to integrate a non-trivial portion of Celestia, just to reuse `nearestStar`. The function depends on so many underlying details, that it is very difficult to separate from Celestia.

2.2. Execution Record/Replay

Execution record and replay is the task of logging and duplicating the execution of a program. The ability to duplicate the execution is valuable in domains such as debugging [71] and security [23], because it helps reveal to programmers the causes behind a program's behavior. The idea behind execution replay is simple: record the instructions as a program executes, and duplicate the instructions later by reading from a log. Most of the instructions are based on deterministic events, such as arithmetic, and can be replayed or re-executed with known inputs. Non-deterministic events pose a key problem, and much research has been devoted to execution replay of different non-deterministic situations in a variety of environments [4, 80, 56].

The record/reuse system Jockey [80] plays an important role in our approach by providing two key services: 1) program checkpointing, and 2) function interception. Checkpointing is the ability

```
const Star* StarBrowser::nearestStar()
{
  Universe* univ = appSim->getUniverse();
  CloserStarPredicate closerPred;
  closerPred.pos = pos;
  std::vector<const Star*>* stars =
     findStars(*(univ->getStarCatalog()),
                 closerPred, 1);
  const Star *star = (*stars)[0];
  delete stars;
  return star;
}
```

Figure 2. From starbrowser.cpp in Celestia.

[†] www.shatters.net/celestia/

DOI: 10.1002/smr

to dump program state to a file at a given time in the program's execution. Interception is the ability to call an arbitrary function whenever a given function executes. Due to space limitations, we direct readers to the related literature for a discussion of these topics [80].

3. OUR APPROACH

Our approach enables the reuse of functions from C and C++ programs. Given a function to reuse, **our approach works in four steps:** 1) from a log file, restore the state of the program containing the function at a point just prior to the function's execution, 2) modify any parameters or global variables as instructed by the programmer, 3) pass control to the function so that it executes, and 4) catch the function return so that the programmer can read the function's output.

In this section, we will elaborate on each of these steps. We will use the example in Figures 3 and 4 to illustrate how these steps work in practice. These figures show how our approach can reuse the function `nearestStar` from Section 2.1.

3.1. Supporting Technology

We have heavily modified the Jockey library [80] for our approach. The most important modification we made was to add the ability to "go live." Many approaches, such as the one implemented in the GNU debugger, do not actually re-execute the logged instructions. Instead, they log the output of each instruction and, during replay, restore the state as it was after the instruction. This restoration produces an identical result when the logs are reviewed for debugging. For our work in reuse, we alter the state before replay, which means that the instructions will need to be re-executed, rather than restored. We implement a "go live" system after the state is restored, inspired by an approach described by Laadan *et al.* [56].

3.2. Preparation

To prepare to reuse a function, a programmer must first record a checkpoint for that function. The checkpoint must be taken at a point just prior to the function's execution. We provide a recording utility based on Jockey's checkpointing feature. The utility takes a program and the name of a function in that program. The utility then executes the program. The programmer may interact with the program to ensure that some behavior is recorded, or run a test script. The utility monitors the process – whenever the function is called, the utility directs Jockey to record the state of the program to a checkpoint file. The function may be called several times, and there will be one checkpoint for each of these. The programmer can choose a checkpoint that he or she prefers, otherwise the default is the first checkpoint.

Each checkpoint corresponds to one function. In order to use multiple functions from one program the user would need to generate a checkpoint for each function. At run time each checkpoint is loaded into its own process. The user is responsible for synchronizing global variables and other external dependencies to insure that the functions behave consistently.

3.3. Reusing Functions

We implemented our approach as a user space C/C++ library for 32-bit Linux 2.6.10. While implementation for different environments is possible, in this paper we limit the scope to one environment for clarity and reproducibility. Figure 3 shows an example program using our library. The remainder of this section will cover the steps of our approach, using this example for context.

3.3.1. Restoring Function State The first step to reuse a function is to restore the program state that lead to the function being called. We use Jockey's checkpoint restoration feature to reload this state from a checkpoint log file. To make this feature available for reuse, we have provided `flashback_load_scene()` in our library (Figure 3, area ①). Like Jockey, a call to this library function will load the program associated with the checkpoint, copy the variable memory space from the log file back into memory, and then skip forward in the program to the point where the

Prepared using smrauth.cls
DOI: 10.1002/smr

```
#include <stdio.h>
#include <stdlib.h>

#include <flashback.h> // access to our library

#include "vecmath.h" // for Celestia's Point3f
#include "star.h" // for Celestia's Star class

int main(int argc, char **argv)
{
    flashback_init();

    struct flashback_scene *s;

(1) s = flashback_load_scene("findstars.scn", 1);

    Point3f new_pos = ...; // set position
    flashback_set_var(s, "pos", new_pos);   (2)

    flashback_exec_scene(s); (3)

(4) void *retval = flashback_get_ret_val(s);
    Star *nearestStar = (Star*)retval;

    return 0;
}
```

Figure 3. Example program using our library to reuse `nearestStar` from Figure 2.

checkpoint was recorded. A key difference between Jockey's default restoration and restoration in our approach is that we use `libdwarf` [26]to place a breakpoint immediately after memory space has been allocated for parameters and other local variables. Libdwarf is a library that allows us to read the DWARF debugging information that is generated by compilers such as GCC and G++ and read by debuggers such as GDB[32]. In this case libdwarf allows us to find the starting address of the target function. Figure 4, area ①, shows an example of where this occurs. The breakpoint causes the program to pause at this location.

Note that the only part of the program executed when loading a scene into memory is the set of instructions between the Jockey Checkpoint and the breakpoint (e.g., the first four instructions in Figure 4). The program is loaded into memory, and skipped forward using data from the checkpoint log file, but no further instructions are dispatched to the processor. The advantage to this technique is that the function is prepared to execute exactly as it was when the checkpoint was recorded: local and global variables are accessible, and dependent functions are available. The programmer does not need to include these details in his or her program. At area ① in Figure 3, the function is ready to be reused. Different class definitions from the program are included for areas ② and ④, however

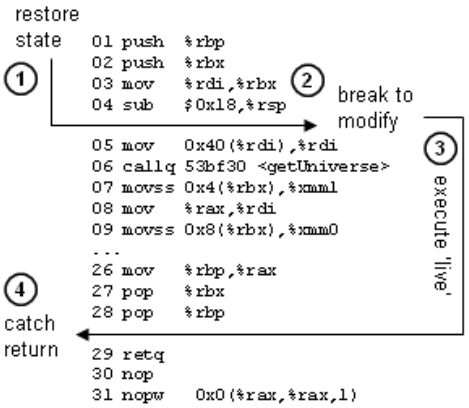


Figure 4. Disassembled `nearestStar` from Figure 2, showing steps taken by the library in Figure 3.

Prepared using smrauth.cls
DOI: 10.1002/smr

these are only necessary to make changes to the execution of the function – they are not required to load or execute the function.

Note as well that a scene contains the entire state of the process including registers, memory, open files, sockets, pipes and other resources. This requires the programmer to be aware of what resources the target function depends on and to update those resources prior to executing the function.

3.3.2. Modifying Local Variables The next step in our approach is to modify local variables. This modification allows programmers to change how the function will execute. For example, in area ② of Figure 3, the value of "pos" in `nearestStar` is changed to an arbitrary position, so that `nearestStar` returns stars near the given arbitrary position. We accomplish this modification in our approach after the breakpoint is reached during restoration (Figure 4, area ②). We access the memory location of the variable, and copy a given variable into that location. Technically, the function `flashback_set_var()` requires the name of the variable and a replacement variable of a compatible type. Our library then uses `libdwarf` to find the variable in memory (and `ptrace` to replace it), so instrumentation is required to find variable addresses by name. Currently, we support getting and setting all primitive types, arrays (including strings), and structs.

At this point the user also has the opportunity to set breakpoints. Breakpoints are defined by a source file name and line number similar to a debugger. This allows the user to execute part of a function and to get or set parameters or local variables possibly by integrating with other functions contained in the host program. The user can then continue the function's execution if they so choose. Otherwise they can unload the scene.

3.3.3. Executing the Function The next step is to execute the function following the breakpoint. When directed by the programmer (Figure 3, area ③), our approach begins executing in two phases. First, we set the program to "live" mode by disabling the replay mechanism in Jockey. We modified Jockey to prevent it from intercepting system calls during "live" mode, so that these calls will be sent to the operating system rather than simulated from the log file. Second, we use `libdwarf` to continue from the breakpoint. The result is that the instructions after the breakpoint will be dispatched to the processor, and the function will execute. The program will behave normally; calls to other functions will cause control to be passed to those functions. Our approach does not affect execution until the function returns. For example, in area ③ of Figure 4, instruction 6 will call `getUniverse`. When `getUniverse` returns, the function will continue from instruction 7 to 28.

3.3.4. Catching Function Return The execution will continue until a breakpoint is hit or until the function is complete. At that time, we catch the return value by intercepting the function's return location, and sending control back to our library rather than to the rest of the program being reused. See area ④ of Figure 4. At this time we are unable to catch non-standard return mechanisms such as setjmp and longjmp. We are also unable to catch calls to custom signal handlers.

f

4. CASE STUDY EVALUATION

This evaluation compares `Flashback` to the standard "copy-paste" method of source code reuse, which is the typical strategy that programmers must follow (see Section 7). In this section, we describe the design of the study, including our research questions, methodology, subjects, and evaluation metrics.

4.1. Research Questions

Our objective is to determine the degree to which `Flashback`, when compared to the copy-paste method, affects the time required by programmers and the number of dependencies that programmers must reuse. To that end, we pose the following Research Questions (RQ):

DOI: 10.1002/smr

RQ_1 To what degree does `Flashback` increase or decrease the time required for programmers to reuse source code?

RQ_2 To what degree does `Flashback` increase or decrease the number of dependencies that programmers must reuse?

RQ_3 To what degree is `Flashback` more or less difficult to use than the copy-paste method?

RQ_4 To what degree is `Flashback` perceived as being more or less appropriate to use than the copy-paste method?

RQ_5 Is `Flashback` perceived as being more applicable to some tasks more than others?

RQ_6 Does `Flashback` affect the running time of a program as compared to the copy-paste method?

The rationale behind RQ_1 is that Flashback is designed to reduce the time required to reuse source code by reducing the number of source-level dependencies. However, `Flashback` introduces its own overhead through the time required to create scenes and debug improper behavior. We have no way to predict whether the time required to successfully apply `Flashback` to a specific case will outweigh the reduction in time gained from eliminating the need to comprehend and integrate source-level dependencies, therefore we investigate it as part of this evaluation.

The rationale behind RQ_2 is that `Flashback` is designed to minimize the number of dependencies needed to reuse code. Even though there is evidence that these dependencies are complex and extensive [38], it is not known precisely how many of these dependencies can be eliminated using `Flashback`. Reducing the number of dependencies benefits programmers during reuse tasks [40], and therefore is a measure of the effectiveness of our tool. We pose RQ_2 to measure the degree of this potential benefit.

The rationale behind RQ_3 is that `Flashback` is designed to replace the copy-paste method. It is not known if the effort required to create and manipulate scenes is outweighed by the effort required to comprehend and integrate a series of source-level dependencies, therefore we investigate it as part of this evaluation.

The rationale behind RQ_4 and RQ_5 is that `Flashback` introduces a new paradigm for code reuse. It is not known if this paradigm serves to make `Flashback` appear more or less appropriate for code reuse, whether generally or for specific tasks. It is also not known if users can relate to using this paradigm so we investigate it as part of this evaluation. It is also not known if `Flashback` introduces any overhead when reusing a function, so we pose RQ_6.

4.2. Methodology

Our methodology for answering our research questions was as follows. We performed a cross-validation study in which programmers completed five code reuse tasks over three rounds. The first round was a calibration task which asked the programmer to reuse a simple function using `Flashback`. This round was not incorporated into the analysis but served to familiarize participants with `Flashback` and give them an opportunity to ask questions without affecting the timings of the subsequent tasks. The remaining two rounds each contained two tasks which asked the participant to reuse the same function from an executable: once through `Flashback` and once through the standard copy/paste method. Table I details the makeup of each round.

We alternated the order of the tasks for each participant as part of the cross-validation design. The purpose of alternating was to insure against a possible bias: without alternating, either method might falsely appear more effective because the participant had already been introduced to the code in the preceding task. The participants were given a questionnaire which asked for the start and end times of each task, the perceived difficulty of the task and any comments they had about the task. At the end of the questionnaire was an exit survey which asked the participant to rate and compare both methods they had employed during the study.

DOI: 10.1002/smr

Round	Task	Subject	Tool/Technique
1	1	*(demo)*	`Flashback`
2	2a	`date`	copy-paste
	2b	`date`	`Flashback`
3	3a	`predict`	`Flashback`
	3b	`predict`	copy-paste

Table I. Cross-validation design of our user study.

4.3. Subject Applications

This section discusses the functions and programs that we ask participants to reuse in the different rounds of our study. The first round involved reusing the `calc_high` function from a demonstration program we wrote called primer. `Calc_high` took three parameters and returned the greatest of those parameters. We asked the participants to change these parameters through `Flashback` and reuse the `calc_high` function in a new program. This round was not included in the analysis but was used to introduce the participant to `Flashback`.

In the second round we asked the participants to reuse the `show_date` function from the Unix utility `date`. This function takes two parameters: a format string and a `timespec` struct defined by the C standard library and containing the number of seconds and microseconds elapsed since the Unix epoch. The function then prints the date on the screen in the format described by the format string. We asked the participants to reuse this function to print the date given some number of seconds elapsed since the Unix epoch. This required the participants to set the value of the `timespec` parameter and read back the return value. In this round we asked the participants to reuse `show_date` both through `Flashback` and the copy/paste method. The function contains 80 lines of code including comments, has a cyclomatic complexity [60] of 13 and calls five functions excluding functions provided by the C library. The enclosing source file depends on ten header files excluding those provided by the standard library. `Show_date` can be exercised by running the date command with no arguments or a valid date string.

The third round asked the participants to reuse functions from a program called `predict`. This program calculates the position of satellites orbiting the Earth given a date. The participants were asked to determine the code in predict relevant to a task, and reuse that code. The task was to calculate the elevation and azimuth of the Moon on January 1, 2000. The participants were asked to do this first using `Flashback` and then with the copy/paste method. Note that `predict` is legacy software which can only be compiled with GCC 2.95 or earlier. Therefore, the programmers needed to port as well as reuse the code from `predict`.

For this task the participants needed to call the `FindMoon` function. This function takes one parameter, a double-precision floating point number representing the time at which the participant is trying to find the position of the moon. The participant must then set a breakpoint at the end of the function in order to get the elevation and azimuth variables. This is necessary because `FindMoon` stores its result in global variables and `Flashback` cannot read global variables. Elevation and azimuth are both double precision floating point numbers. To exercise `FindMoon` the participants only need to press "l" from the main screen. The `FindMoon` function is 168 lines long including comments and has a cyclomatic complexity of 3. It calls two functions not present in the standard library and depends on five variables defined outside the function. The enclosing source file depends on no header files other than those provided by the standard library.

4.4. Participants

We recruited eight programmers to participate in our study. These programmers were graduate students with an average of 10 years programming and 8.6 years programming in C or C++. All but one participant had experience using the GNU debugger (GDB).

DOI: 10.1002/smr

#	Question
1	Please rate the difficulty of the task.
2	I found the method I used to be appropriate for this task.
3	I can see why someone would use this method for this task.
4	I can see why someone would use this method for a different task.
5	Comments about the method you used?
6	Please rate the difficulty of using flashback overall.
7	Please rate the difficulty of using the copy-paste method overall.
8	For the date tasks, which was easier (select 4 for about the same).
9	For the predict tasks, which was easier (select 4 for about the same).
10	In general, what was the biggest problem you encountered for either method.
11	What would you say are the biggest strengths and weaknesses of flashback.
12	Please give any other detailed comments you have.

Table II. Questions asked as part of our user study. We asked questions 1 through 5 after each task, but 6 through 12 only after the participant completed all tasks.

4.5. Evaluation Metrics and Tests

We asked each participant to fill out a questionnaire as part of the user study. Table II lists the questions. Each question had a multiple-choice Likert-Scale answer that ranged from 1 (strongly disagree) to 7 (strongly agree), except for questions 5, 10, 11, and 12, which were open-ended. We also collected start and end times for each task as well as comments about each task and the study in general. To determine whether the differences between the means of the Likert-scale scores were significant, we used a Mann-Whitney statistical test [85]. The Mann-Whitney test is appropriate for this analysis because 1) it is non-parametric and we cannot test that our data are normally distributed because of the small sample size, and 2) it is unpaired and we do not always have equal numbers of responses from all participants (e.g., some participants skipped questions).

We answered each research question by evaluating the answers to the questionnaire as well as the programs that the participants wrote. We answered RQ_1 by evaluating the total time taken for each task. We answered RQ_2 by evaluating the line count and cyclomatic complexityof each program. Line count has been shown [35] to broadly predict the fault-proneness of a module and cyclomatic complexity provides a more detailed picture. We used the Pmccabe program[‡] to evaluate the cyclomatic complexity of the source files submitted by the participants. RQ_3 drew from questions 1, 6 and 7 in Table II. RQ_4 drew from questions 2, 3 and 4 in Table II. RQ_5 drew from questions 8 and 9 in Table II. Finally, we answered RQ_6 by timing the correct implementations of task 2 and the one correct submission for task 3. We compared the running time of the `Flashback` version of the program to the running time of the copy-paste version of the same program. The program pairs for task 2 were run for a total of three thousand times. The single correct pair for task 3 was run three thousand times as well.

[‡]https://packages.debian.org/sid/pmccabe

Prepared using smrauth.cls
DOI: 10.1002/smr

4.6. Threats To Validity

As with any study, our evaluation carries threats to validity. We identified two main sources of threats. First, our evaluation was conducted by human experts, each of whom spent approximately 4 hours on our evaluation. The results could have been influenced by participant fatigue or variations in programming experience. For each round the participant was asked to reuse the same code using two different methods. Whichever method was used last could have unduly benefited from the knowledge of the code gained through the first method. Moreover participants did not always answer every question in the survey. Finally, the questions we asked the participants could create experimenter bias [20] where the participant supports our approach because he knows it is what we want. We attempted to mitigate these threats through our cross-validation design which rotated the order of the tasks in each round. This assured that each method was equally exposed to participant fatigue as well as the benefits of the participant having previously seen the code. We also recruited our participants from a diverse body of students and confirmed our results with accepted statistical testing procedures. We removed all pairs of questions where the participant answered the question concerning one method but left it blank concerning the other. Still, we cannot guarantee that a different group of participants would not produce a different result.

The second source of threats is the set of applications we chose. We only reused code from two applications. We attempted to mitigate this threat by choosing programs with differing levels of complexity. We also attempted to mitigate this threat by choosing applications that together required the full set of features present in `Flashback`. In the case of `predict`, we attempted to choose a real-world case that would illustrate the conditions under which we expect `Flashback` to be the most useful: an application with code that cannot be compiled with a modern compiler without a great deal of effort. We ran the entire evaluation inside a virtual machine including RQ_6 in order to avoid any issues stemming from running a 2003-era operating system in a modern environment. This also prevented us from needing to configure any of the participants' computers to support `Flashback`. Neither of the applications stressed the system which prevents us from getting a complete picture of possible bottlenecks and performance issues arising from `Flashback`. We cannot guarantee that a different set of applications would not produce a different result.

5. EVALUATION RESULTS

In this section, we present the results of the evaluation of our approach. We report our empirical evidence behind, and answers to, RQ1 and RQ2.

5.1. RQ1: Time to Reuse Source Code

We found statistically-significant evidence that programmers required less time to complete the programming tasks when using `Flashback` as compared to the copy-paste method. The average time taken to reuse a function through `Flashback` was 35 minutes whereas the average time for the copy/paste method was 47 minutes. Figure 5(a) shows a statistical summary of the times required. To determine statistical significance, we used the Wilcoxon signed-rank test by posing hypothesis $\mathrm{H1}_0$:

H1_0 The difference between the time taken to reuse a function through `Flashback` and the time taken to reuse a function through the copy/paste method is not statistically significant.

We rejected hypothesis $\mathrm{H1}_0$ based on the results of the Wilcoxon test shown in Table III. An α value of 0.05 means that the difference between the samples was statistically significant, meaning that the programmers required less time when using `Flashback` by a statistically significant margin. This holds true despite the fact that one participant was able to complete the copy-paste `Predict` task more quickly because of prior experience (see section 5.7).

5.2. RQ2: Program size and dependencies

We found statistically-significant evidence that programs that used `Flashback` had fewer lines of code and had a smaller cyclomatic complexity compared to the copy-paste method. The average line

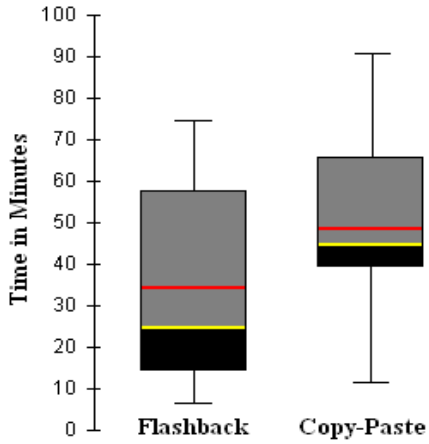
DOI: 10.1002/smr

count for `Flashback` was 32 lines and the average cyclomatic complexity was 1.3 whereas the average line count for the copy/paste method was 197 lines and the average cyclomatic complexity was 16.75. Figure 5(b) shows a statistical summary of the code sizes. To determine statistical significance, we used the Wilcoxon signed-rank test by posing hypotheses $H2_0$ and $H3_0$:

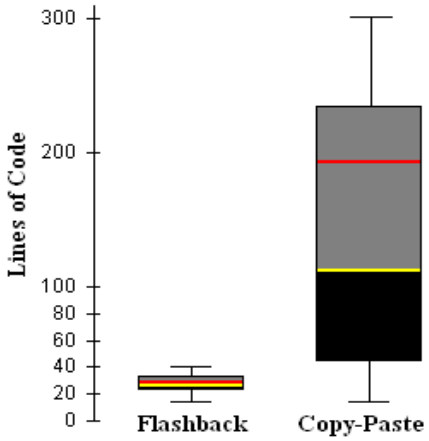
H2_0 The difference between the number of lines of code for a program using `Flashback` and the number of lines of code for the same program using the copy/paste method is not statistically significant.

H3_0 The difference between the cyclomatic complexity for a program using `Flashback` and the cyclomatic complexity for the same program using the copy/paste method is not statistically significant.

We rejected hypotheses $H2_0$ and $H3_0$based on the results of the Wilcoxon test shown in Table III. An α value of 0.05 means that the difference between the samples was statistically significant, meaning that the programmers produced code with fewer dependencies when using `Flashback` by a statistically significant margin. The larger variance in the line count and cyclomatic complexity for the copy-paste method highlights the different code reuse strategies the participant employed.



(a)



(b)

Figure 5. Boxplots comparing (a) time in minutes and (b) lines of code required to complete programming tasks using `Flashback` and copy-paste. The red line indicates the mean and the yellow line separating the gray and black areas indicates the median

DOI: 10.1002/smr

Some used a greedy strategy that tried to satisfy the dependencies whereas others tried to use as little of the code as possible.

5.3. RQ3: Perceived Ease-of-use

We found statistically-significant evidence that `Flashback` was perceived to be easier to use than the copy-paste method, as recorded by the participants' responses to Q1, Q6 and Q7 in the user study (see Table II). To determine statistical significance of the differences in these responses, we used the Wilcoxon signed-rank test by posing hypotheses $H4_0$ and $H5_0$:

H4_0 Generally, the difference between the ease-of-use rating for `Flashback` and the ease-of-use rating for the same program using the copy/paste method is not statistically significant.

H5_0 For each round, the difference between the ease-of-use rating for `Flashback` and the ease-of-use rating for the same program using the copy/paste method is not statistically significant.

We rejected hypotheses $H4_0$ and $H5_0$ based on the results of the Wilcoxon test shown in Table III. H4 was based on Q1 which asked the participants to rate the difficulty of the method they had used to complete each task. H5 was based on Q6 and Q7. Q6 asked the participants to rate `Flashback` overall and Q7 did the same for the copy-paste method. The lower variance of H4 suggests that the participants were more certain about the two methods after they had completed the study. The lower variance of the copy-paste method in both H3 and H4 suggests that in both cases the participants were more certain of the scores they gave the copy-paste method.

5.4. RQ4: Perceived Appropriateness

We found statistically-significant evidence that `Flashback` was perceived to be more appropriate and relatable than the copy-paste method, as recorded by the participants' responses to Q3 and Q4 in the user study (see Table II). To determine statistical significance of the differences in these responses, we used the Wilcoxon signed-rank test by posing hypotheses $H6_0$, $H7_0$ and $H8_0$:

H6_0 For each round, the difference between the appropriateness rating for `Flashback` and the appropriateness rating for the same program using the copy/paste method is not statistically significant.

H7_0 Generally, the difference between the relatability rating for `Flashback` and the relatability rating for the same program using the copy/paste method is not statistically significant.

H8_0 For each round, the difference between the relatability rating for `Flashback` and the relatability rating for the same program using the copy/paste method is not statistically significant.

We rejected hypotheses $H6_0$, $H7_0$ and $H8_0$ based on the results of the Wilcoxon test shown in Table III. H6 was based on Q2 which asked the participants to rate the appropriateness of the method they had used for each task. H7 and H8 were based on Q3 and Q4. Q3 asked the participants to rate how easily they could relate the method they had just used to the task they had just completed and Q4 asked the participants to rate how easily they could relate the method they had just used to other tasks in general. The larger variance across all three hypotheses for the copy-paste method suggests that certain participants were more comfortable with the copy-paste method in general. Programmers who are used to reading and adapting code may be more likely to prefer to continue to use familiar techniques.

5.5. RQ5: Perceived Ease-of-Use for Certain Tasks

We found statistically-significant evidence that there are tasks for which `Flashback` was perceived to be more appropriate than the copy-paste method, as recorded by the participants' responses to Q8 and Q9 in the user study (see Table II). To determine statistical significance of the differences in these responses, we used the Wilcoxon signed-rank test by posing hypothesis $H9_0$:

DOI: 10.1002/smr

H9_0 The difference between the ease-of-use rating for the `Date` tasks and the `Predict` tasks is not statistically significant.

We rejected hypothesis H9_0 based on the results of the Wilcoxon test shown in Table III. An α value of 0.05 means that the difference between the samples was statistically significant, meaning that the programmers reported that `Flashback` was easier to use for certain tasks by a statistically significant margin. H9 was based on Q8 and Q9, which asked the participant to rate whether `Flashback` or the copy-paste method was easier to use for the `Date` and `Predict` tasks respectively.

5.6. RQ6: Running Times

We found statistically-significant evidence that `Flashback` increases the running time of a program as compared to the copy-paste method. The average running time of a program using `Flashback` was 44 milliseconds whereas the average running time of a program that used the copy-paste method was one millisecond. To determine statistical significance of this difference we used the Wilcoxon signed-rank test by posing hypothesis H10_0:

H10_0 The difference between the running times for the `Flashback` programs and the copy-paste programs is not statistically significant..

We rejected H10_0 hypothesis based on the results of the Wilcoxon test shown in Table III. An α value of 0.05 means that the difference between the samples was statistically significant, meaning that `Flashback` increases the running time of a program by a statistically significant margin. It is important to note that neither the programs or the variables involved stressed the system in any way meaning that this analysis cannot give us a detailed picture of the performance issues a programmer would encounter using `Flashback`.

5.7. Qualitative Results

Participants in the evaluation study had the opportunity to provide feedback on how well the methods they used worked for each task as well as generally (see questions 5, 10, 11 and 12 in Table II). In this section we explore these opinions for feedback on our approach and directions for future work.

Participants found that task 2 was made more complex by the large number of external dependencies despite being simple in principal:

> The dependencies are killing me. Date.c wouldn't actually compile because it is missing includes that it calls. A lot of the missing stuff I was able to hack around but the hacks are terrible. I wasn't able to finish in an hour. If I wrote it from scratch I would have been done by now.

> Implementation Notes: Copied `show_date`. Found 2 static variables I needed to also copy from compile errors. Needed to define `DATE_FMT_LANGINFO`, not sure if it should be this `nl_languinfo(_DATE_FMT)`, would need to research to ensure correctness. Needed to include time.h stdint.h, locale.h. Used simpler fprintf for error reporting. May not be "correct". Had to download coreutils to see def of nstrftime and x2nrelloc. Replaced with equiv. I included a screen dump of calls to make in task2q/make.history. Comments: Copying source like this caused numerous header/dependency issues so I tend to reimplement a specific solution using std UNIX/POSIC functions. This exercise really drives home how annoying copying foreign "generic" source is. It is frustrating converting these internal lib calls such as x2urealloc to equiv. POSIC code. Clarification on (5): I would have decided to write my own solution upon seeing GNU's implementation.

> I could not complete the task for following reasons: 1) `x2nrealloc` is not declared in any file under src/, 2) `LC_all` is not declared in any file under src/, 3) `setlocale` is not declared in any file under src/, 4) `nstrftime` is not declared in any file under src/

DOI: 10.1002/smr

The crux of the `show_date` method boils down to the `ustrftime` function. I could not locate this header, so I switched to the `strftime` function after copying the whole `show_date` function. As a result, I used copy/paste to lead to a stripped down version that uses a different function call. The downside: This is not a robust method on the level with `show_date`.

The participants found `Flashback` to be easier for this task:

This was much easier to do. My only troubles were with passing indirect params but I just made silly mistakes with casting. This was also easier because I tried copy-paste before.

As with task 1, I created the snapshot w/o issue. I got confused by incorrectly setting format when parameters to pointers in `pretty_date.c` (code commented). Apparently I must change existing structures/strings in the snapshot only? What if the string parameter in the snapshot is a string literal and not a char buffer? Overall this worked as expected and implemented w/o any issue! Only a small confusion on the parameter API as noted above, I took 8 minutes to solve.

Yes. `Flashback` is very helpful as it just reuses the runtime functionality without having to get into the compilation troubles.

In contrast, the difference between the two methods was not as pronounced for task 3:

Copy-paste worked fairly well here. There were some dependencies to take care of but it worked correctly in the end, unlike the other method which I was unable to figure out.

copy-paste would be no less effective, at least than using flashback, since code reusable for this task is easy to extract and there is no complex dependencies.

... This actually wasn't a horrendous copy/paste task as that seems to be the way the code was written in the first place (copied from Javascript).

Participants reported that in general dependencies complicated their experiences with the copy-paste method and that `Flashback` did not require them to use those dependencies.

"Copy-paste: complex external (even internal) dependencies make reusable even unfeasible. I failed to reuse the `Date` task anyway within an hour."

"Copy-paste: too many dependencies, often impossible to include them all. Flashback: N/A"

"The biggest strength [of `Flashback`] is definitely its ability to use compiled code without all the dependencies and source code bloat that comes with using those sources."

The most common criticism of `Flashback` was that it could be made more developer-friendly by simplifying the API and developing a means of debugging programs inside `Flashback`. Another common criticism was `Flashback`'s inability to access global variables. A selection of these comments follows.

"`Flashback` uses a new paradigm which will require rethinking some programming methods. The global variable problem is frustrating."

"I think this is a really interesting idea, it just needs a little work making it more developer friendly. For example, when the wrong checkpoint file is specified, the resulting error message is not at all indicative of the actual 'file not found' error."

"Strengths: no issues with dependant functions, no issue with cross-compiling. Weakness: global variable access, lack of generalizable errors (infinite loops!)"

"For copy-paste, the biggest problem was resolving the dependencies in the code. For flashback, it was debugging."

DOI: 10.1002/smr

"Difficult to debug. There is no control over target program."

Some participants encountered strange errors with the `Flashback` tasks that could impact the time taken to complete the task. At least one participant was able to complete a copy/paste task more quickly because of experience prior to the study.

> The idea is great. I can borrow the `FindMoon` function from `Predict` without needing to figure out the weird state information. I still cannot get the output shown in the doc. My process: 1) Run `Predict` through `Flashback` using Now as the time, 2) Load the scene, set daynum = epoch day, 3) Run the scene, 4) Extract el and az from the scene, 5) Display el and az.

> It seemed to work except reading the result at the end. It appears as though it was just reading random memory because the result was always the same regardless of input. Furthermore, the `Flashback` method seems next to impossible to debug.

> I copy-pasted `FindMoon`. I got several make errors to fix: KRR C parameter typing and C++. Included numerous missing globals. Math.h / math defs copied. Took observer info out of predict/predict.qth and hardcoded qth global. Copied missing helper functions. Changed def of `FindMoon` to set az, el double pointers. Removed code after these are set. Result is the same as with using flashback. Other I found it suprisingly straightforward to copypaste the code with few issues, non serious. Finding the right code was easier based on T and A experience.

One participant noted that "I think this would be great for nontechnical programmers (scientists who code, biologists, chemists). If it was implemented in a simpler language python, perl, R, it would allow them to reuse fast C code in an interpreter environment. I also wonder how this would work with pthreads or MPI or even GPL. While this can help technical people reuse code and the same time it can enable nontechnical people to use code they could never write on their own." The same participant also noted that "`Flashback` allows you to preserve the optimizations the original programmer created. My copy-paste `Date` code was a memory leaking hack. For larger programs this is unreasonable and `Flashback` would be excellent. Flashbacks weakness would be that it puts too much overhead on that program. I want to call, break, get, and set without too much management."

6. DISCUSSION

Our paper advances the state-of-the-art by proposing a new strategy for code reuse based upon program record and replay. Additionally, we have developed `Flashback` as an implementation of this strategy. We have made the `Flashback` source code as well as our study results available online to promote independent research [§].

Our evaluation has shown that programmers take significantly less time to reuse a function using `Flashback` as opposed to the copy-paste method. We also showed that the resulting code is simpler when using `Flashback` to reuse a function. Participants found `Flashback` to be easier than the copy-paste method and expressed optimism about the possibility of integrating `Flashback` into their own projects. Our evaluation does not compare `Flashback` to other tools that assist the programmer in transplanting code. Examples include `Gilligan`, developed by Holmes and Walker [39] and `Skipper`, developed by Makady and Walker [58]. These tools improve on the traditional copy-paste method in different ways; `Skipper` uses the existing test suite to test the newly transplanted code where as `Gilligan` simplifies the process of transplanting additional source code by walking the programmer through the additional source code and keeping track of the reuse plan. `Gilligan` and `Skipper` both operate on Java code where as `Flashback`

[§] `http://cse.nd.edu/~aarmaly/papers/flashback`

DOI: 10.1002/smr

operates on C and C++ code. This makes a direct comparison impossible at this time. Still, there are scenarios where `Flashback` offers unique benefits.

One benefit of `Flashback` is the reuse of legacy code that is tightly bound to a specific environment, e.g. a specific compiler, compiler version, specific versions of external libraries, etc. Migrating code between these kinds of environments requires the programmer to understand complex changes in the environment that can span years. An inadequate understanding of these changes can introduce subtle bugs that are difficult to track down. These problems are further compounded when the code is specific to a domain with which the programmer is not familiar. Another benefit of `Flashback` is the extraction functions that are tightly bound to the surrounding code. In cases where transplanting a large amount of additional source code is impractical or undesirable `Flashback` offers an easy way to integrate the required functionality, provided the programmer can identify the appropriate function.

6.1. Limitations

The limitations of our approach can be classified into two groups: those limitations unique to `Flashback` and those limitations that are independent of implementation. At this time `Flashback` is unable to recognize or manipulate C++ objects and templates. `Flashback` also depends on the Jockey library for its record and replay support. Jockey depends on Linux kernel 2.6 which constrains `Flashback` to older Linux distributions. A Jockey replacement would need to be capable of saving a program's state at a specified point and then restoring the state. The program state would need to include not only the heap and stack but anonymous memory mappings commonly used instead of the heap for large allocations. Any Jockey replacement would also have to contend with address space layout randomization (ASLR) which causes variable locations to be different for each run of the program. For purposes of our study we disable ASLR but this is not practical in the field as ASLR is a security feature.

`Flashback` also depends on version 2 of the DWARF standard. Deploying `Flashback` on a modern system would require it to be capable of supporting the changes introduced by versions 3 and 4 of DWARF. `Flashback` is also unable to read the contents of global variables at this time. As Libdwarf is capable of locating global variables implementing this functionality is a matter of time and effort. The dependence on DWARF means that `Flashback` cannot read programs that have been stripped of debugging symbols. Any alternatives to DWARF would need to be capable of locating functions and variables, some of which cannot be precisely located until runtime.

`Flashback` complicates debugging by having the target function run in its own process. This has the advantage that a program that uses `Flashback` can reuse code from a target program that also uses `Flashback`. In order to debug the target function a debugger would not only need to take into account the fact that the child process is running a completely separate program, but would need to compensate for the fact that some addresses of variables have changed because of the presence of Jockey. Finally, `Flashback` scenes contain the entire kernel state including open files, pipes, sockets, memory mappings and other resources. The user will need to reinitialize any external resources that the target function depends on. Since these dependencies are not always easy to find or comprehend their presence can further complecate both the use and debugging of a `Flashback` scene.

Using record and replay for code reuse introduces several limitations irrespective of the implementation. The most basic requirement is for the target function to exist in the first place. If there is no function that it its present form satisfies the programmer's requirements then the programmer is forced to either try to adapt an existing function or create a new one. The programmer still needs to read the source code to understand the order, types and meaning of the function parameters. Determining the type of parameters becomes more complicated when they are not base types. The programmer must then determine whether they are simple type definitions of base types or if they are complex structures. Complex structures often reference other structures, forcing the programmer to read them to determine whether they are relevant to the task at hand. After this is done the programmer still must include that portion of the source code that defines any types that are not base types. The programmer must also be aware of any operating system resources the

DOI: 10.1002/smr

function requires such as file descriptors, sockets, pipes, or shared memory. These will need to be manually initialized to new values by the host program. In many cases these resources are passed as parameters to the function making them readily apparent but in some cases they may be defined as global variables or as the return value of some other function called by the target function. All of this work must be done when using record and replay or the copy/paste method.

In cases where the target program needs to be built in an environment different than the host environment record and replay has the potential to save time and effort in porting the program to the host environment. However, any runtime dependencies must also be ported. In cases where they cannot simply be compiled in e.g. a program requiring specific older versions of the kernel or of the C library, this could present extra barriers to deployment. Maintenance and testing become more complex because the developers have to either contend with a binary blob or a part of the source code that must be built separately from the rest. In these cases it may become simpler to comprehend and integrate the relevant source code of the target program including its dependencies.

6.2. Performance Issues

In this section we discuss two aspects of `Flashback` that have the potential to introduce execution overhead compared to the copy-paste method. The first is the time taken to load a scene. A scene contains all memory allocated by the target process including the heap, stack and any memory-mapped files. In the case of complex programs or programs that deal with large amounts of data this can cause the scene to become extremely large. The entire scene must be read from disk every time the user wishes to execute the target function in order to reliably reset the state of the target process. If a function must be called many times and the containing scene is extremely large the invocation of the target function will take longer as compared to the copy-paste method because the scene must be read from disk each time.

The second aspect is the copying of large variables. `Flashback` uses the `ptrace` system call to copy data to and from the child process. `Ptrace` copies data four bytes at a time. The data is copied to kernel space before being copied to the user space of the receiving process. As a result ptrace is slower than copying the same amount of memory locally. The copy-paste method does not require the programmer to copy the contents of variables since the target function is part of the same process. As a result, functions that process large variables have the potential to run slower when using `Flashback` as compared to the copy-paste method. Examples include compression or searching of large files as well as performing operations on large matrices. Modern versions of Linux offer the `process_vm_readv` and `process_vm_writev` functions that can move data of arbitrary size directly from the user space of the sending process to the user space of the receiver.

6.3. Future Work

Future work will focus on three areas. First, we will eliminate the dependency on Jockey which constrains `Flashback` to legacy versions of Linux. Alternatives to Jockey need not capture the complete program state as Jockey does. Capturing the state of the target program's memory and registers will suffice. Other components of program state such as open files or network connections will change when loading a scene into a new program. Second, we will reduce `flashback`'s learning curve by simplifying the interface. `Flashback` currently requires the programmer to use one of three families of functions to get and set variables. The correct behavior can be determined automatically through Dwarf debugging information. Additionally, `Flashback` cannot access global variables so the programmer must capture the variables that a global variable is derived from; Libdwarf can already locate global variables so implementing them is a matter of time and effort. We will also explore rerouting dependencies, meaning that some dependent functions in the target program will not be called but instead control will return to the host program which will call a replacement function. Finally, we will compare `Flashback` to code reuse tools other than the traditional copy-paste method. Possible comparisons include Gilligan [39] and Skipper [58] which are static approaches. It remains to be seen whether `Flashback` can outperform these approaches and whether certain cases lend themselves to one approach over the others.

DOI: 10.1002/smr

7. RELATED WORK

A diverse body of research has emerged in the area of source code reuse. While the benefits of reuse are widely recognized, including higher quality due to pre-tested code [92, 49] and reduced time of development [6, 5], there is a rich and varied set of opinions and approaches to reuse [28]. Some aim to reuse entire frameworks, models, and high-level designs. Others point to reuse at a low-level, including classes, functions, and even individual processor instructions. However, in general software reuse research flows from search and retrieval of reusable code from repositories, to automated systems for transplanting that code, and to approaches that extract the code from the project where it originates. We survey each of these areas in this section.

7.1. Automated Reuse Systems

Different strategies have been proposed for reusing source code found in software repositories. At the highest level, techniques for product-line engineering guide the entire development process, ensuring that many products and built using similar methodology to reduce incompatability among those products, in effect maximizing the opportunities for reuse [27, 95, 47, 2, 33, 73, 64, 22]. The second author's own work, Prefab [62], falls into this category by recommending source code that implements features common to many software products. Nevertheless, much of the reuse in product-line engineering involves source code that was never intended to be reused, a problem known as "pragmatic" [39, 37], "opportunistic" [36, 11], or even "scrapheap" reuse [52]. Approaches in this arena, in particular work by Holmes *et al.* [39], help programmers plan for reuse by prioritizing changes needed to transplant source code. Makady and Walker [58] proposed a solution called Skipper that uses the existing program's test suite to test whether the transplanted code still functions as it did. A key difference from our work is that we aim to minimize the number of changes that are needed, hiding the unnecessary details by using execution information.

7.2. Software Decomposition

Much research has focused on the problem of decomposing software into different reusable sections. The premise is that each section is responsible for a different set of features, and that to reuse one feature, only one section of the software is necessary. For example, program *slicing* is a technique to locate all statements needed for a given statement [84, 93, 94, 9, 96]. If a programmer wanted to reuse a section of a program, he or she could create a "slice" based on that section. The slice would contain all other parts of the program that supported the section to be reused. Slicing tends to follow one of two strategies. First, *static* slicing identifies all candidate dependencies of a given statement, relying purely on analysis of the source code [94, 41, 59, 91, 53]. The static approach tends to create very large slices, between 30 and 60% of the program [8]. *Dynamic* approaches have been proposed as a solution by only including statements actually executed [1, 50, 51, 69]. There are techniques that create slices by mixing static and dynamic information [54, 14, 65, 31], to prune slices to reduce their size [99, 86], and to assign probabilistic weights to different statements [100].

Slicing has a wide range of applications [29, 77], but still requires programmers to understand many implementation details that are included in the slice, if the slice is to be reused. Another approach is to look for components which show signs of being reusable. Though reuse is largely subjective [82], some signs these approaches look for are low coupling [25], independent subgraphs [98], and cyclomatic complexity [13]. A key recent development has been the reuse of code from programs *during execution*. Return-oriented programming, including the well-publicized "Frankenstein" prototype [66], demonstrates how arbitrary programs can be constructed by mining reusable "gadgets" from program executables [78, 15, 16, 10]. Our proposed work is related to these approaches in that we break down existing projects into reusable sections, but is different in that we do not rely on the programmer to create a blueprint of the new program to be generated. Instead we help programmers to reuse elements of program state directly from the execution logs.

7.3. Execution Record and Replay

Execution record and replay is the task of logging the execution of a program, so that the execution can be repeated. The ability to repeat the execution is valuable in domains such as debugging [71] and security [23], because it helps reveal to programmers the causes behind a program's behavior.

DOI: 10.1002/smr

The idea behind execution replay is simple: record the instructions as a program executes, and repeat the instructions later by reading from a log [32]. Most of the instructions are based on deterministic events, such as arithmetic, and can be replayed or re-executed with known inputs.

Non-deterministic events pose a key problem, and much research has been devoted to execution replay of different non-deterministic situations in a variety of environments. One solution to non-deterministic replay is to record the state of every variable for every instruction, and load that state during replay [57, 4, 75]. While effective, this solution imposes a high performance penalty in terms of execution speed and the size of the execution logs [72]. To reduce this penalty, some approaches focus on recording only that final few seconds of a program in order to catch a failure [97, 88]. Another direction has been to record only certain sections of the execution that are necessary for playback. Different methods have been proposed, including hardware support [3, 21, 42, 67, 68, 70, 71, 74], virtualization [12, 23, 24, 18, 55], application-level logging [30, 34, 80, 74], programming language features [57, 79, 17], and operating system modifications [7, 87, 89, 56]. These approaches have been successful, but carry certain tradeoffs. For example, hardware support requires special equipment that may not be widely available, and some situations, such as race conditions, must be recorded entirely [63, 81].

In building record and replay for our research, it is important to support the ability to "go live." Many approaches, such as the one implemented in the GNU debugger [32], do not actually re-execute the logged instructions. Instead, they log the output of each instruction and, during replay, restore the state as it was after the instruction. This restoration produces an identical result when the logs are reviewed for debugging. This approach is referred to as 'omniscient" or "back-in-time" debugging. To reduce overhead, omniscient debuggers such as TOD [76] store events that allow it to reconstruct the program state at any given point rather than storing the entire program state every time something changes. Others such as TimeMachine [¶] make use of hardware support available on some processors to avoid introducing overhead entirely.

For our work in reuse, we alter the state before replay, which means that the instructions need to be re-executed, rather than restored. Approaches do exist for altering the execution of a program for testing [90] and that support "going live" after the state is restored, so that all following instructions are re-executed [56]. We build on these approaches, specifically work by Laadan *et al.* [56] that introduced rendezvous and sync points. Using these points, we can selectively restore or re-execute different sections of the log, depending on what variables or conditions the programmer wishes to alter. For example, in some cases, a system call can simply be restored, if the results from that call are irrelevant to the final outcome of replay. In other cases, several system calls need to be re-executed (e.g., network or file access). Full details of our system are in section 3.

8. CONCLUSION

We have presented a new technique for pragmatic source code reuse. Our approach uses execution record and replay technology, from the area of software debugging, to capture the state of a program prior to the execution of a target function. We then restore the state and let the program continue in order to execute the target function inside a host program. In an empirical study, we found that when compared to the copy-paste method, the size of reused programs was reduced by up to a factor of 6, and that time to completion was reduced by a statistically-significant margin. Our study demonstrates that execution record/replay is a viable alternative to manual copy-paste source code reuse. It remains to be scene whether execution record/replay is more effective than other tools that aim to simplify pragmatic code reuse.

[¶] http://www.ghs.com/products/timemachine.html

Prepared using smrauth.cls
DOI: 10.1002/smr

ACKNOWLEDGMENTS

We thank and acknowledge the eight participants in our study for their time and helpful feedback.

REFERENCES

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.
2. G. Arango and R. Prieto-Diaz. *Domain Analysis: Acquisition of Reusable Information for Software Construction*. IEEE Comp. Society Press, May 1989.
3. D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, PADD '91, pages 194–206, New York, NY, USA, 1991. ACM.
4. R. M. Balzer. Exdams: extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM.
5. R. D. Banker and R. J. Kauffman. Reuse and productivity in integrated computer-aided software engineering: an empirical study. *MIS Q.*, 15(3):375–401, Oct. 1991.
6. B. Barns and T. Bollinger. Making reuse cost-effective. *Software, IEEE*, 8(1):13 –24, jan. 1991.
7. P. Bergheaud, D. Subhraveti, and M. Vertes. Fault tolerance in multiprocessor systems via application cloning. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 21–, Washington, DC, USA, 2007. IEEE Computer Society.
8. D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2), Apr. 2007.
9. D. Binkley, M. Harman, and J. Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.
10. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.
11. J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.
12. T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, Feb. 1996.
13. G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, Feb. 1991.
14. G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information and Software Technology*, 40(1112):595 – 607, 1998.
15. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.
16. P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient detection of the return-oriented programming malicious code. In *Proceedings of the 6th international conference on Information systems security*, ICISS'10, pages 140–155, Berlin, Heidelberg, 2010. Springer-Verlag.
17. J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.
18. J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with crosscut. *SIGPLAN Not.*, 45(7):13–24, Mar. 2010.
19. J. W. Davison, D. M. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, pages 44–54, 2000.
20. N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies. Yours is better!: participant response bias in hci. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1321–1330. ACM, 2012.
21. J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.
22. H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *ICSE*, pages 181–190, 2011.
23. G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, Dec. 2002.
24. G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM.
25. M. F. Dunn and J. C. Knight. Automating the detection of reusable parts in existing software. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 381–390, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
26. M. Eager. The dwarf debugging standard, Apr. 2012. http://www.dwarfstd.org/.
27. W. Frakes, R. Prieto-Diaz, and C. Fox. Dare: Domain analysis and reuse environment. *Annals of Software Eng.*, 5, 1998.

DOI: 10.1002/smr

28. W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, July 2005.
29. K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, Aug. 1991.
30. D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.
31. D. Giffhorn and C. Hammer. Precise slicing of concurrent programs. *Automated Software Engg.*, 16(2):197–234, June 2009.
32. GNU. The gnu project debugger, Sept. 2012. http://www.gnu.org/software/gdb/.
33. H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
34. Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.
35. T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.
36. B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.
37. R. Holmes, T. Ratchford, M. P. Robillard, and R. J. Walker. Automatically recommending triage decisions for pragmatic reuse tasks. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 397–408, Washington, DC, USA, 2009. IEEE Computer Society.
38. R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 447–457, Washington, DC, USA, 2007. IEEE Computer Society.
39. R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology*, 2012. To appear.
40. R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1–20:44, Feb. 2013.
41. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.
42. D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.
43. S. Islam, J. Krinke, D. Binkley, and M. Harman. Coherent clusters in source code. *Journal of Systems and Software*, 88:1–24, 2014.
44. R. E. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1988.
45. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.
46. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, Mar. 2007.
47. K. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58 – 65, jul/aug 2002.
48. C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.
49. J. C. Knight and M. F. Dunn. Software quality through domain-driven certification. *Ann. Softw. Eng.*, 5:293–315, Jan. 1998.
50. B. Korel. Computation of dynamic program slices for unstructured programs. *IEEE Trans. Softw. Eng.*, 23(1):17–34, Jan. 1997.
51. B. Korel and J. Rilling. Dynamic program slicing methods. *Information & Software Technology*, 40(11-12):647–659, 1998.
52. G. Kotonya, S. Lock, and J. Mariani. Opportunistic reuse: Lessons from scrapheap software development. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 302–309, Berlin, Heidelberg, 2008. Springer-Verlag.
53. J. Krinke. Static slicing of threaded programs. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '98, pages 35–42, New York, NY, USA, 1998. ACM.
54. J. Krinke. Effects of context on program slicing. *J. Syst. Softw.*, 79(9):1249–1260, Sept. 2006.
55. O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. Dejaview: a personal virtual computer recorder. *SIGOPS Oper. Syst. Rev.*, 41(6):279–292, Oct. 2007.
56. O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *SIGMETRICS Perform. Eval. Rev.*, 38(1):155–166, June 2010.
57. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, Apr. 1987.
58. S. Makady and R. J. Walker. Validating pragmatic reuse tasks by leveraging existing test suites. *Software: Practice and Experience*, 43(9):1039–1070, 2013.

DOI: 10.1002/smr

59. B. Malenfant. Flow analysis to detect blocked statements. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 62–, Washington, DC, USA, 2001. IEEE Computer Society.
60. T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.
61. M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.
62. C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 848–858, Piscataway, NJ, USA, 2012. IEEE Press.
63. S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):235–244, Apr. 1991.
64. M.Moon, K.Yeom, and H.S.Chae. An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *TSE*, 31, 2005.
65. M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Program slicing with dynamic points-to sets. *IEEE Trans. Softw. Eng.*, 31(8):657–678, Aug. 2005.
66. V. Mohan and K. W. Hamlen. Frankenstein: Stitching malware from benign binaries. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, Bellevue, Washington, August 2012.
67. P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.
68. P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 73–84, New York, NY, USA, 2009. ACM.
69. G. B. Mund and R. Mall. An efficient interprocedural dynamic slicing method. *J. Syst. Softw.*, 79(6):791–806, June 2006.
70. S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. *SIGOPS Oper. Syst. Rev.*, 40(5):229–240, Oct. 2006.
71. S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, May 2005.
72. R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, PADD '93, pages 1–11, New York, NY, USA, 1993. ACM.
73. N.Niu and S.Easterbrook. Extracting and modeling product line functional requirements. *RE*, 2008.
74. M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, Mar. 2009.
75. D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, PADD '88, pages 124–129, New York, NY, USA, 1988. ACM.
76. G. Pothier and É. Tanter. Back to the future: Omniscient debugging. *Software, IEEE*, 26(6):78–85, 2009.
77. M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 406–416, New York, NY, USA, 2002. ACM.
78. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.
79. M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. *SIGPLAN Not.*, 31(5):258–266, May 1996.
80. Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 69–76, New York, NY, USA, 2005. ACM.
81. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, Oct. 1997.
82. S. R. Schach and X. Yang. Metrics for targeting candidates for reuse: an experimental approach. In *Proceedings of the 1995 ACM symposium on Applied computing*, SAC '95, pages 379–383, New York, NY, USA, 1995. ACM.
83. J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, July 2008.
84. J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.
85. M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *CIKM*, pages 623–632, 2007.
86. M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.
87. S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.
88. D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, pages 109–120, New York, NY, USA, 2011. ACM.
89. H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, pages 265–272, Washington, DC, USA, 2000. IEEE Computer Society.
90. P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*,

DOI: 10.1002/smr

ICST '11, pages 200–209, Washington, DC, USA, 2011. IEEE Computer Society.
91. G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 107–119, New York, NY, USA, 1991. ACM.
92. G. von Krogh, S. Spaeth, and S. Haefliger. Knowledge reuse in open source software: An exploratory study of 15 open source projects. *Hawaii International Conference on System Sciences*, 7:198b, 2005.
93. M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
94. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
95. D. M. Weiss and C. T. R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
96. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.
97. M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.
98. L. Zhang, J. Luo, H. Li, J. Sun, and H. Mei. A biting-down approach to hierarchical decomposition of object-oriented systems based on structure analysis. *J. Softw. Maint. Evol.*, 22(8):567–596, Dec. 2010.
99. X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 169–180, New York, NY, USA, 2006. ACM.
100. Y. Zhang and R. Santelices. Predicting runtime data dependences for slice inspection prioritization. Technical Report 2012-02, University of Notre Dame Computer Science Department, July 2012.

DOI: 10.1002/smr

Table III. Statistical summary of the results for RQ_1 through RQ_4. Wilcoxon test values are T, T_{expt}, and T_{vari}. Decision criteria are Z, Z_{crit}, and p.

RQ	Metric	H	Approach	Samples	$\tilde{x}$	μ	Vari.	T	T_{expt}	T_{vari}	Z	Z_{crit}	p
RQ_1	Minutes	H_1	`Flashback` copy-paste	15 15	25.00 45.00	34.80 48.67	558.5 431.1	29.00	59.50	309.5	1.734	1.645	0.041
RQ_2	L.O.C.	H_2	`Flashback` copy-paste	15 15	32 118	32 197.4	53.07 54410	6.00	67.50	373.6	3.182	1.645	0.001
RQ_2	CC.	H_3	`Flashback` copy-paste	16 16	1.313 16.750	0.363 780.333	1.000 8.000	0.000	65.000	369.250	-3.383	1.960	0.001
RQ_3	Likert Scale	H_4	`Flashback` copy-paste	13 5.0	3.00 5.154	3.154	3.474 2.808	12.000	45.500	199.000	-2.375	-1.645	0.009
RQ_3	Likert Scale	H_5	`Flashback` copy-paste	8 5.0	3.00 5.250	3.250	1.929 1.357	3.000	17.500	49.750	-2.056	-1.645	0.020
RQ_4	Likert Scale	H_6	`Flashback` copy-paste	14 5.50	1.00 5.071	1.857	1.978 3.764	5.00	52.50	252.250	-2.991	-1.645	0.001
RQ_4	Likert Scale	H_7	`Flashback` copy-paste	15 5.0	1.00 4.600	1.933	2.495 4.543	6.000	60.000	306.625	-3.084	-1.645	0.001
RQ_4	Likert Scale	H_8	`Flashback` copy-paste	15 2.000	2.0 3.533	2.133	2.124 4.552	21.0	58.5	303.625	-2.152	-1.645	0.016
RQ_5	Likert Scale	H_9	`date` predict	6 2.0	7.0 4.0	6.5	0.700 5.2	18.0	10.0	22.250	1.696	1.645	0.045
RQ_6	seconds	H_{10}	`Flashback` predict	6000 0.001	0.044 0.000	0	0.045 0.001	18003000.000	9001500.000	17517132691.250	68.012	1.960	<0.0001

DOI: 10.1002/smr