Reusable Execution Replay: Execution Record and Replay for Source Code Reuse

Ameer Armaly, Casey Ferris, and Collin McMillan
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{aarmaly, cferris1, cmc}@nd.edu

ABSTRACT

A key problem during source code reuse is that, to reuse even a small section of code from a program, a programmer must include a huge amount of dependency source code from elsewhere in the same program. These dependencies are notoriously large and complex, and many can only be known 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 a preliminary study in which two programmers used our tool to complete four tasks over four hours.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Algorithms, Design

Keywords

Source code reuse, execution record and replay

1. INTRODUCTION

Source code reuse has long been a centerpiece of software engineering research [16]. This research has yielded many effective solutions for designing reusable software, such as object-oriented architectures [10] and repositories of shared libraries [12]. While these technologies have proliferated, the reality is that much source code is not designed with reuse in mind [9]. As a result, *pragmatic* reuse (also called "copy-paste" or "opportunistic" reuse [14]) has become an accepted practice in many professional environments [13, 11]. 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 dependency source code from elsewhere in the same project. The complexity and size of these dependencies (typically totaling 30% to 60% of the original program [2]) forces programmers to choose: either cut the reused code to reduce dependency, or abandon the reuse altogether [9]. This difficulty of understanding dependencies is a consistent theme in studies of software reuse [20, 6].

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 [15, 19]. 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.

We have implemented our technique as a library for reusing functions from C/C++ programs in a Linux environment. Using our library, a programmer can select a function in a program that he or she would like to reuse. Then, the programmer executes the program. Our tool records the state of the program any time that control is passed to the function. To reuse the function, the programmer directs the library to reload that state at a given point in the new program. The programmer can then alter the state as necessary, such as the values of the function's arguments. At runtime, the library makes these changes, restores the state, and passes control to the function. Once the function is complete, our library passes control is 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.

Our work is an exploratory solution to a classic problem of source code reuse. We present a preliminary study comparing our solution to the typical "copy-paste" manual reuse approach. We recruited two programmers to complete four programming tasks each in a total of four hours. Both programmers found our tool to be very effective at reducing the amount of dependency source code that the programmers needed to read, understand, and include.

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 1, 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 these data types and dependencies 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 [17] and security [7], 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 [1, 19, 15].

The record/reuse system Jockey [19] plays an important role in our approach by providing two key services: 1) program checkpointing, and 2) function interception. Checkpointing is the ability 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 [19].

```
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 1: From starbrowser.cpp in Celestia.

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 2 and 3 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 [19] 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.* [15].

3.2 Preparation

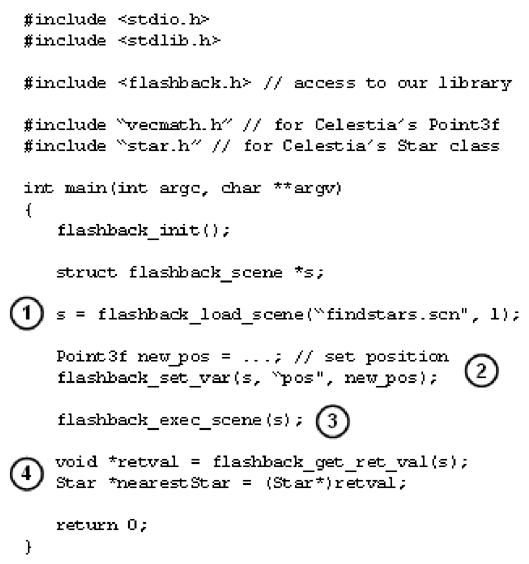
To prepare to reuse a function, a programmer must first record a checkpoint for that function. The checkpoint must be taken at a point just prior to the function's execution. We provide a recording utility based on Jockey's checkpointing feature. The utility takes a program and the name of a function 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.

3.3 Reusing Functions

We implemented our approach as a userspace 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 2 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

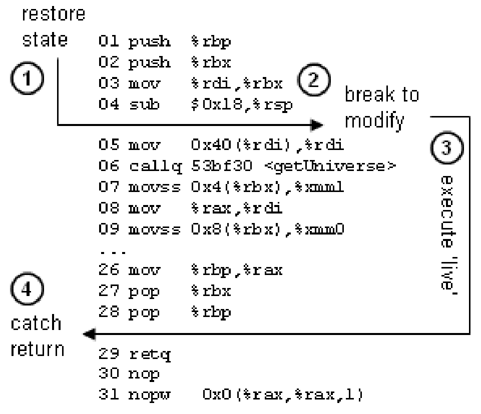


```
#include <stdio.h>
#include <stdlib.h>

#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;

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

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

   flashback_exec_scene(s);

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

   return 0;
}
```

Figure 2: Example program using our library to reuse `nearestStar` from Figure 1.

reuse, we have provided `flashback_load_scene()` in our library (Figure 2, 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 checkpoint was recorded. A key difference between Jockey's default restoration and restoration in our approach is that we use `libdwarf` [8], a debugging library, to place a breakpoint immediately after memory space has been allocated for parameters and other local variables. Figure 3, 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 is the set of instructions between restoration and the breakpoint (e.g., the first four instructions in Figure 3). 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 dependency functions are available. The programmer does not need to include these details in his or her program. At area 1 in Figure 2, the function is ready to



```
01 push  %rbp
02 push  %rbx
03 mov   %rdi,%rbx
04 sub   $0x18,%rsp
05 mov   0x40(%rdi),%rdi
06 callq 53bf30 <getUniverse>
07 movss 0x4(%rbx),%xmm1
08 mov   %rax,%rdi
09 movss 0x8(%rbx),%xmm0
...
26 mov   %rbp,%rax
27 pop   %rbx
28 pop   %rbp
29 retq
30 nop
31 nopw   0x0(%rax,%rax,1)
```

Figure 3: Disassembled `nearestStar` from Figure 1, showing steps taken by the library in Figure 2.

be reused. Different classes from the program are included for areas 2 and 4, however these are only necessary to make changes to the execution of the function – they are not required to load or execute 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 2, 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 3, 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.

3.3.3 Executing the Function

The next step is to execute the function following the breakpoint. When directed by the programmer (Figure 2, 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 3, 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 3.

4. PRELIMINARY EVALUATION

We conducted a preliminary evaluation in which we studied the following research questions:

RQ$_1$ Does our technique reduce the number of dependencies that programmers must reuse?

RQ$_2$ Does our technique reduce the time required for programmers to reuse source code?

Our methodology for answering these questions was to conduct a user study in which programmers completed programming tasks using our tool. To compare our tool with a competitive approach, the programmers also completed tasks using a "copy-paste" strategy, which is the typical strategy that programmers must follow [9]. In a copy-paste strategy, programmers read the code to reuse, copy that code into their own programs, then either copy in any dependency code, or modify the reused code so that it does not need the dependencies.

We asked the programmers to complete four tasks. The first task was to use the copy-paste method to reuse source code from the Linux utility `date` to translate a given Unix epoch time into a human-readable time format (e.g., 946702800 into 12:00am Jan 1, 2000 EST). The second task was to reuse the same code using our tool. The third task was to use our tool to write a program that calculates the position of Earth's moon at a given time. The programmers reused code from `predict`, a satellite location calculator. For the fourth task, the programmers used the copy-paste method to implement the same functionality as in the third task. Note that `predict` is a legacy program which does not compile in GCC versions newer than 2.95. Therefore, tasks 3 and 4 required the programmers to port before reusing it.

Two programmers participated in our study and shed light on our research questions. For RQ_1, both programmers created substantially smaller programs using our tool than the copy-paste method. These programs were smaller because they did not include many of dependencies required for the copy-paste method. For tasks 1 and 2 (reusing `date`), the copy-pasted program was 117 LOC for one programmer and 110 LOC for the other, compared to 38 LOC and 29 LOC when using our tool. Likewise, for tasks 3 and 4, the copy-pasted program was 236 LOC for one programmer compared to 37 LOC when using our tool. The other programmer could not finish the copy-pasted `predict` code, but wrote a 32 LOC solution using our tool. Both programmers expressed frustration during the copy-paste method, writing "these dependencies are killing me" and "copying source like this caused numerous header/dependency issues."

For RQ_2, we observed a drop in the amount of time required to complete the tasks when using our tool. One programmer finished the `date` task in 40 minutes using copy-paste, versus 16 minutes with our tool. The other programmer finished the same task in 66 minutes compared to 25 minutes with our tool. On tasks 3 and 4, one programmer was unable to finish using the copy-paste method, but took 40 minutes with our tool. The other programmer finished copy-paste in 12 minutes versus 25 minutes for our tool – the only instance in which our tool took longer to use than the copy-paste method. Nevertheless, the programmer stated that this was only possible due to prior experience with similar code.

5. RELATED WORK

There are three key areas of related work: execution record and replay, pragmatic reuse, and return-oriented programming. Execution record/replay is described in Section 2.2. Pragmatic reuse tools have been studied extensively by Holmes *et al.* [9]. These tools differ from our work in that the previous tools help programmers plan reuse tasks by identifying dependencies and recommending modifications. Our work focuses on eliminating the need to transplant many of these dependencies. Return-oriented programming is a technique from computer security in which malicious software reuse "gadgets" from existing programs to confuse anti-virus software [18, 4, 5, 3]. One key difference in our work is that we do not depend on blueprints of assembly instructions.

6. CONCLUSION

We have presented a new technique for source code reuse. Our approach uses execution record and replay technology, from the area of software debugging, to capture the dependencies of source code from a program's execution. The state of these dependencies is then made available for reuse. In a preliminary study, we found that the size of reused programs was reduced by **up to a factor of 6**, and that time to completion was reduced by at least **half** in 3 of 4 instances.

7. REFERENCES

[1] R. M. Balzer. Exdams: extendable debugging and monitoring system. In *AFIPS*, pages 567–580, 1969.
[2] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM TOSEM*, 16(2), Apr. 2007.
[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ASIACCS*, 2011.
[4] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *17th ACM CCS*, 2010.
[5] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient detection of the return-oriented programming malicious code. In *6th ICISS*, 2010.
[6] J. W. Davison, D. M. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Tech. Journal*, pages 44–54, 2000.
[7] 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.
[8] M. Eager. The dwarf debugging standard, Apr. 2012. http://www.dwarfstd.org/.
[9] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM TOSEM*, 21(4):20:1–20:44, Feb. 2013.
[10] R. E. Johnson and B. Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1988.
[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *31st ICSE*, 2009.
[12] 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.
[13] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *13th WCRE*, 2006.
[14] G. Kotonya, S. Lock, and J. Mariani. Opportunistic reuse: Lessons from scrapheap software development. In *Proceedings of the 11th CBSE*, 2008.
[15] 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.
[16] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.
[17] 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.
[18] 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.
[19] Y. Saito. Jockey: a user-space library for record-replay debugging. In *6th AADEBUG*, 2005.
[20] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE TSE*, 34(4):434–451, July 2008.