Language-Based Replay via Data Flow Cut

Ming Wu Fan Long[†] Xi Wang[*] Zhilei Xu[*] Haoxiang Lin Xuezheng Liu
Zhenyu Guo Huayang Guo[†] Lidong Zhou Zheng Zhang
Microsoft Research Asia [†]Tsinghua University [*]MIT CSAIL
{miw,v-falon,haoxlin,xueliu,zhenyug,v-huguo,lidongz,zzhang}@microsoft.com
{xi,timxu}@csail.mit.edu

ABSTRACT

A replay tool aiming to reproduce a program's execution interposes itself at an appropriate replay interface between the program and the environment. During recording, it logs all non-deterministic side effects passing through the interface from the environment and feeds them back during replay. The replay interface is critical for correctness and recording overhead of replay tools.

iTarget is a novel replay tool that uses programming language techniques to automatically seek a replay interface that both ensures correctness and minimizes recording overhead. It performs static analysis to extract data flows, estimates their recording costs via dynamic profiling, computes an optimal replay interface that minimizes the recording overhead, and instruments the program accordingly for interposition. Experimental results show that iTarget can successfully replay complex C programs, including Apache web server and Berkeley DB, and that it can reduce the log size by up to two orders of magnitude and slowdown by up to 50%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Languages, Performance, Reliability

Keywords

Data flow, graph cut, replay, instrumentation

1. INTRODUCTION

A replay tool aims at reproducing a program's execution, which enables cyclic debugging [28] and comprehensive diagnosis techniques, such as intrusion analysis [7, 13], predicate checking [9, 17], program slicing [40], model checking [14, 39], and test generation [8, 24, 30].

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

Re-execution of a program could often deviate from the original execution due to *non-determinism* from the environment, such as time, user input, and network I/O activities. A replay tool therefore interposes at an appropriate *replay interface* between the program and the environment, recording in a log all non-determinism that arises during execution. Traditional choices of replay interfaces include virtual machines [7], system calls [33], and higher-level APIs [10, 11]. For correctness, at the replay interface the tool *must* observe all non-determinism during recording, and eliminate the non-deterministic effects during replay, e.g., by feeding back recorded values from the log. Furthermore, both interposition and logging introduce performance overhead to a program's execution during recording; it is of practical importance for a replay tool to minimize such overhead, especially when the program is part of a deployed production system.

This paper proposes iTarget, a replay tool that makes use of programming language techniques to find a correct and low-overhead replay interface. iTarget achieves a replay of a program's execution with respect to a given *replay target*, i.e., the part of the program to be replayed, by ensuring that the behavior of the replay target during replay is identical to that in the original execution. To this end, iTarget analyzes the source code and instruments the program during compilation, to produce a single binary executable that is able to run in either recording or replay mode.

Ensuring correctness while reducing recording overhead is challenging for a replay tool. Consider the Apache HTTP Server shown in Figure 1, consisting of a number of plug-in modules that extend its functionality. The server communicates intensively with the environment, such as clients, memory-mapped files, and a database server. The programmer is developing a plug-in module `mod_X`, which is loaded into the Apache process at runtime. Unfortunately, `mod_X` occasionally crashes at run time. The programmer's debugging goal is to reproduce the execution of replay target `mod_X` using iTarget and inspect suspicious control flows.

The first challenge facing iTarget is that it must interpose at a *complete* replay interface that observes all non-determinism. For example, the replay target `mod_X` may both issue system calls that return non-deterministic results, and retrieve the contents of memory-mapped files by dereferencing pointers. To replay `mod_X`, iTarget thus must capture non-determinism that comes from both function calls and direct memory accesses. An incomplete replay interface such as one composed of only functions [10, 11, 26, 31, 33] would result in a failed replay. A complete interposition at an instruction-level replay interface observes all non-determinism [2], but it often comes with a prohibitively high interposition overhead, because the execution of each memory access instruction is inspected.

Another challenge is that iTarget should choose a replay interface *wisely* and prefer one with a low recording overhead. For example,

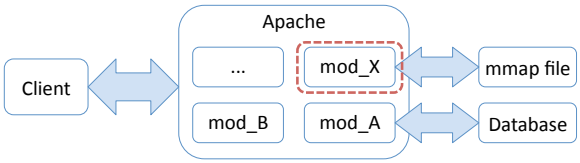


Figure 1: The Apache HTTP Server process consisting of several modules communicates with the environment. The Apache module `mod_X` (enclosed by the dash line) is the replay target.

if `mod_X`'s own logic does not directly involve database communications, it should be safe to ignore most of the database input data during recording for replaying `mod_X`. Naively recording all input to the *whole* process [10, 11] would lead to a huge log size and significant slowdown. However, if `mod_X` is tightly coupled with `mod_B`, i.e., they exchange a large amount of data, it is better to replay both modules together rather than `mod_X` alone, so as to avoid recording their communications.

iTarget addresses these challenges with the benefit of language-based techniques for replay. First, iTarget instruments a program at the granularity of instructions (in the form of an intermediate representation used by compilers) for the interposition at the replay interface. Such a fine granularity is *necessary* for correctly replaying programs with sources of non-determinism from non-function interfaces, e.g., memory-mapped files.

In addition, iTarget models a program's execution as a *data flow graph*; data flows across a replay interface are directly correlated with the amount of data to be recorded. Therefore, the problem of finding the replay interface with a minimal recording overhead is reduced to that of finding the minimum cut in the data flow graph. In doing so iTarget instruments a needed *part* of the program and records data accordingly, which brings down the overhead of both interposition and logging at runtime. The actual interposition is through compile-time instrumentation at the chosen replay interface as the result of static analysis, thereby avoiding the execution-time cost of inspecting every instruction execution.

We have implemented iTarget on Windows x86, and applied it to a wide range of C programs, including Apache HTTP Server, Berkeley DB, two HTTP clients neon and Wget, and a set of SPEC CINT2000 (the integer component of SPEC CPU2000) benchmarks. Experimental results show that iTarget can reduce log sizes by up to two orders of magnitude and reduce performance overhead by up to 50% when some logical subset of a program is chosen as a replay target. Even when the whole program is chosen as the replay target, iTarget's recording performance is still comparable to that of state-of-the-art replay tools.

The contributions of this paper are twofold: 1) a data flow model for understanding and optimizing replay tools, and 2) a language-based replay tool that provides both a high correctness assurance and a low overhead.

The rest of the paper is organized as follows. Section 2 presents a replay model. Section 3 describes how iTarget computes a replay interface via static analysis. Section 4 describes iTarget's runtime for recording and replay. Section 5 discusses the choice of a replay target in practice. We evaluate iTarget in Section 6, survey related work in Section 7, and then conclude in Section 8.

2. MODEL

The foundation of iTarget hinges on our replay model, which provides a general framework for understanding replay correctness, as well as the associated recording overhead. The model naturally explains different strategies of existing replay tools and paves the way for iTarget's language-based replay approach. Both the replay model and different replay strategies with respect to the model are the subject of this section.

```
f() {                                  // execution
  cnt = 0;                             1 cnt1 <- 0
  g(&cnt); printf("%d\n", cnt);        2 a1 <- random()
  g(&cnt); printf("%d\n", cnt);        3 cnt2 <- cnt1 + a1
}                                      4 print cnt2
g(int *p) {                            5 a2 <- random()
  a = random(); *p += a;               6 cnt3 <- cnt2 + a2
}                                      7 print cnt3
```

Figure 2: A code snippet and its execution.

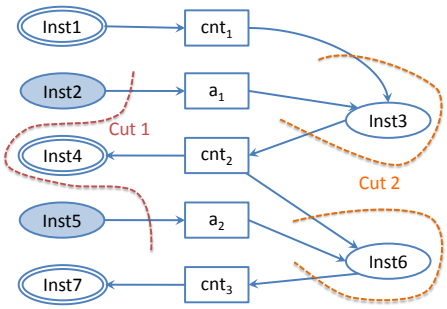


Figure 3: Execution flow graph. Ovals represent operation nodes and rectangles represent value nodes. Of all operation nodes, double ovals are target operations for replay, while shadow ovals are non-deterministic operations.

2.1 Execution Flow Graph

We first assume single-threaded executions; multi-threading issues will be discussed in Section 2.3.

We use the code listed in Figure 2 as a running example, where function f calls function g twice to increase a counter by a random number. Each variable in the execution is attached with a subscript indicating its version, which is bumped every time the variable is assigned a value, such as $cnt_{1,2,3}$ and $a_{1,2}$. The seven instructions in the execution sequence are labeled as $Inst_{1-7}$.

We model an execution of a program as an *execution flow graph* that captures data flow, as illustrated in Figure 3. An execution flow graph is a bipartite graph, consisting of *operation nodes* (represented by ovals) and *value nodes* (represented by rectangles). An operation node corresponds to an execution of an instruction, while the adjacent value nodes serve as its input and output data. Each operation node may have several input and output value nodes, connected by *read* and *write* edges, respectively. For example, $Inst_3$ reads from both cnt_1 and a_1, and writes to cnt_2. A value node is identified by a variable with its version number; the node may have multiple read edges, but only one write edge, for which the version number is bumped. Each edge is weighted by the volume of data that flow through it (omitted in Figure 3).

An execution flow graph covers the code either written by the programmer or adopted from supporting libraries. The programmer can choose part of code of her interest as the replay target; a replay target corresponds to a subset of operation nodes, referred to as *target nodes* (represented by double ovals), in an execution flow graph. For example, to replay function f in Figure 3, $Inst_{1,4,7}$ are set as target nodes. The *goal* of replay is to reproduce an identical run of these target nodes, defined as follows.

DEFINITION 1. *A replay with respect to a replay target is a run that reproduces a subgraph containing all target nodes of the execution flow graph, as well as their input and output value nodes.*

The programmer can also choose a subset of value nodes as the replay target. Since an execution flow graph is bipartite, it is equivalent to choose their adjacent operation nodes as the replay target. We assume that the replay target is a subset of operation nodes in the following discussion.

A naive way to reproduce a subgraph is to record executions of *all* target nodes with their input and output values, but this will likely introduce a significant and unnecessary overhead. One way to cope with this is to take advantage of *deterministic* operation nodes, which can be re-executed with the same input values to generate the same output. For example, assignments (e.g., $Inst_1$) and numerical computations (e.g., $Inst_3$) are deterministic. In contrast, *non-deterministic* operation nodes correspond to the execution of instructions that generate random numbers or receive input from the network. These instructions cannot be re-executed during replay, because each run may produce a different output, even with the same input values. In Figure 3, non-deterministic operation nodes are represented by shadow ovals (e.g., $Inst_{2,5}$).

Since a non-deterministic node cannot be re-executed, to ensure correctness a replay tool can record either the *output* of that non-deterministic operation node, or the *input* of any deterministic operation node that is affected by the output of that non-deterministic operation node, and feed the recorded values back during replay.

To replay target nodes correctly, a replay tool must ensure that target nodes are not affected by non-deterministic nodes, as manifested as a path from a non-deterministic operation node to any of the target nodes. A replay tool can introduce a *cut* through that path, like Cuts 1 and 2 given in Figure 3. Such a cut is called a *replay interface*, defined as follows.

DEFINITION 2. *Given an execution flow graph, any graph cut that partitions non-deterministic operation nodes from target nodes gives a valid replay interface.*

A replay interface partitions operation nodes in an execution flow graph into two sets. The set containing target nodes is called the *replay space*, and the other set containing non-deterministic operation nodes is called the *non-replay space*. During replay, only operation nodes in replay space will be re-executed.

A replay tool should log the data that flow from non-replay space to replay space, i.e., through the cut-set edges of the replay interface, because the data are non-deterministic. Recall that each edge is weighted with the cost of the corresponding read/write operation. To reduce recording overhead, an optimal interface can be computed as the minimum cut [4], defined as follows.

DEFINITION 3. *Given an execution flow graph, the minimum log size required to record the execution for replay is the maximum flow of the graph passing from the non-deterministic operation nodes to the target nodes; the minimum cut gives the corresponding replay interface.*

2.2 Cut Strategies

One simple strategy for finding a replay interface is to cut non-determinism *eagerly* whenever any surfaces during execution by recording the output values of that instruction. Take Figure 3 as an example. Given non-deterministic operation *random*, Cut 1 prevents the return values of $Inst_{2,5}$ from flowing into the rest of the execution. A replay tool that adopts this strategy will record the values that flow through the edges $(Inst_2, a_1)$ and $(Inst_5, a_2)$; in this case, $Inst_{2,5}$ are in non-replay space while the rest of the nodes are in replay space.

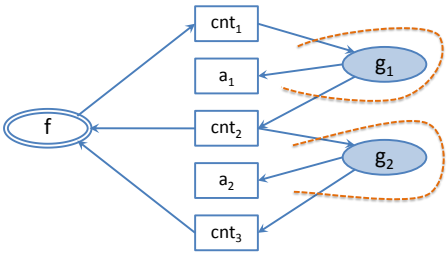


Figure 4: Condensed execution flow graph. The cut corresponds to Cut 2 in Figure 3.

Function-level cut. We can impose an additional cut constraint that instructions of the same function will be either re-executed or skipped *entirely*, i.e., a function as a whole belongs to either replay space or non-replay space. A function-level cut provides a natural debugging unit for the programmer and avoids switching back and forth between replay and non-replay spaces within a function. We believe that such a function-level cut offers better debugging experience in practice, even though it may lead to a slightly larger log. iTarget computes a function-level cut.

For a function-level cut, iTarget condenses instructions in an execution of a function into a single operation node. As shown in Figure 4, g_1 (including $Inst_{2,3}$) and g_2 (including $Inst_{5,6}$) are two calls to function g, which returns a non-deterministic value. The cut in Figure 4 corresponds to Cut 2 in Figure 3. Note that this cut also employs the *eager* strategy, which tries to cut non-determinism by recording the output whenever an execution of a function involves non-deterministic operation nodes. A replay tool that adopts the strategy will record the values that flow through the edges (g_1, cnt_2) and (g_2, cnt_3); in this case, $g_{1,2}$ and $a_{1,2}$ are in non-replay space while the rest of the nodes are in replay space.

Neither of the two "eager" cuts shown in Figure 3 and 4 is optimal, because some of the returned non-deterministic values may never be used by their callers and thus can be stripped during recording; previous replay tools [2, 10, 11, 26, 31, 33] generally use similar eager strategies. Another simple cut strategy is to wrap precisely around the replay targets in replay space [24], leaving the rest in non-replay space, which is usually not optimal either (see Section 6.3). iTarget employs a lazy, globally-optimized strategy based on the minimum cut, which can result in smaller recording overhead (see Section 3 for details).

2.3 Multithreading

Thread interleaving introduces another source of non-determinism that may change from recording to replay. For example, suppose threads t_1 and t_2 writes to the same memory address in order in the original run. A replay tool must enforce the same write order during replay; otherwise, the value at the memory address may be different and the replay run may diverge from the original one.

To reproduce the same run, a replay tool should generally record information of the original run in two kinds of logs: a *data flow log* as defined in our model, and a *synchronization log* with regard to thread interleaving. We discuss recording strategies for producing the synchronization log as follows.

The first strategy is to record the *complete* information about how thread scheduling occurs in the original run. The tool can either 1) serialize the execution so that only one thread is allowed to run in the replay space, or 2) track the causal dependence between concur-

rent threads enforced by synchronization primitives (e.g., locks). The two methods are standard techniques used by many runtime debugging tools [10, 11, 22]. Note that causal dependence tracking may be used along with a race detector [32, 36] that eliminates unprotected memory accesses beforehand.

The second strategy, on the other extreme, is to record *nothing* in the synchronization log, assuming a deterministic multithreading model [5, 23]. In this case the thread scheduler behaves deterministically, so that the scheduling order in the replay run will be the same as that in the original run. The replay tool can then use the data flow log alone to reproduce the replay run.

iTarget supports both recording strategies for multithreaded programs. Note that the strategy for producing the synchronization log is orthogonal to the data flow log.

3. REPLAY INTERFACE COMPUTATION

Although the minimum cut in an execution flow graph precisely defines the replay interface with the minimum log, as described in Section 2, the cut is only optimal with respect to that *specific* run, and is known only *after* the run. iTarget instead estimates a general replay interface that approximates the optimal one *beforehand* and *statically*. This section describes how iTarget constructs a data flow graph via static analysis and finds a cut as the replay interface.

3.1 Static Flow Graph

To approximate execution flow graphs statically, iTarget computes a *static* flow graph of a program via program analysis to estimate the execution flow graphs of all runs. For example, because version information of both value nodes and operation nodes may be only available during run-time rather than during compile-time, $cnt_{1,2,3}$ in the execution flow graph (Figure 3) may be projected to a single value node cnt in a static flow graph; similarly, g_1 and g_2 in Figure 4 may be projected into a single operation node g. The weight of each edge is given via runtime profiling under typical workloads (see Section 3.3). The minimum cut of the resulting static flow graph is computed as the recommended replay interface, which is expected to approximate the optimal ones in typical runs.

A static flow graph can be regarded as an approximation of the corresponding execution flow graphs, where operation nodes are functions and value nodes are variables. The approximation should be *sound*: a cut in the static flow graph should correspond to a cut in the execution flow graph.

iTarget performs static analysis to construct a static flow graph from source code, as follows.

First, iTarget scans the whole program and adds an operation node for each function and a value node for each variable (in the SSA form [29]).

Secondly, iTarget interprets each instruction as a series of reads and writes. For example, $y = x + 1$ can be interpreted as `read` x and `write` y. Every time iTarget discovers a function f reading from variable x, it adds an edge from x to f; similarly, it adds an edge from f to y if function f writes to variable y.

Finally, iTarget performs pointer analysis and determines variable pairs that may *alias*, i.e., they may represent the same memory address, and merges such pairs into single value nodes. Specifically, iTarget uses a classical Andersen-style pointer analysis [1]. The analysis is flow- and context-insensitive, which means that it does not consider the order of statements (though it uses the SSA form to partially do so) nor different calling contexts in a program. In this way, the analysis is both efficient and correct for multithreaded programs.

As a result, a static flow graph that iTarget constructs can be considered as a projection from an execution flow graph: invocations of the same functions are merged into single operation nodes, and variables that may alias are merged into single value nodes. Thus, the approximation is sound, as a cut in a static flow graph is easily translated to one in a corresponding execution flow graph. We omit the proof detail for brevity.

3.2 Missing Functions

The analysis for constructing a static flow graph requires the source code of all functions. For functions without source code, such as low-level system and libc calls, iTarget speculates their side effects as follows.

By default, iTarget conservatively considers functions without source code as non-deterministic, i.e., they are placed in non-replay space. Consequently, these functions are not re-executed during replay, so iTarget must record their side effects. iTarget assumes that such functions will modify memory addresses reachable from their parameters. For example, for function `recv`$(fd, buf, len, flags)$, iTarget assumes that `recv` may modify memory reachable from buf. As a result, iTarget would cut at all the read edges that flow from variables affected by buf to the replay space.

The default approach also works smoothly with non-deterministic functions involving global variables. For example, the global variable `errno` is defined internally in libc and may be modified by a libc function without source code (considered as non-deterministic); When `errno` is read by the program, iTarget will place the corresponding value node of `errno` in non-replay space. Thus, the replay interface may cut through the read edges of the value node. iTarget can then log its value during recording and feed the value back during replay.

A downside of the default approach is that, if a program calls `recv` once to fill buf and then reads the content ten times, iTarget would have to record ten copies of buf. To reduce recording overhead further, iTarget reuses R2's annotations [11] on 445 Windows API functions to complete a static flow graph. For example, R2 annotates function `recv` with buf that will be modified and with the size of the buffer. iTarget exploits the knowledge to fix the static flow graph with a write edge from `recv` to buf; it may then choose to cut at the write edge and record only one copy of buf.

In addition, iTarget annotates dozens of popular libc functions as *deterministic*, including math functions (e.g., `abs`, `sqrt`), memory and string operations (e.g., `memcpy`, `strcat`), since they do not interact with the environment. iTarget can place them in replay space to avoid recording their side effects if possible.

It is worth noting that iTarget uses function annotations *optionally*, for reducing recording overhead rather than for correctness. iTarget does not require any annotations for higher-level functions. These annotations are also shared across different applications and do not require programmer involvement.

3.3 Minimum Cut

iTarget weights each edge in the static flow graph with the volume of data that pass through it. Unlike edges in a dynamic execution flow graph, the weight of each edge in a static flow graph depends on the number of invocations of the corresponding instructions. iTarget estimates the weight via runtime profiling. Given a replay target in a weighted static flow graph, iTarget computes the minimum cut using Dinic's algorithm [6]. It runs in $O(|V|^2|E|)$ time, where $|V|$ and $|E|$ are the numbers of vertices and edges in the static flow graph, respectively.

Profiling can be done in a variety of ways; for example, by using an instruction-level simulator or through sampling. Currently, iTarget simply builds a profiling version of a program, where memory access instructions are all instrumented to count a total size of data

transfers with each of them. We run this version multiple times on a sample input. For functions that are never invoked during profiling, iTarget assigns a minimal weight to their corresponding edges.

Generally, the recording overhead may depend on the extent to which the profiling run reflects the execution path of the actual recording run. Different types of workload in profiling and recording runs may drive the runs to different execution paths and hence negatively affect the recording performance. However, in our experience, the resulting cut tends *not* to be sensitive to the profiling workload scale (see Section 6.4). Thus, the programmer can profile the program with smaller workload scale without incurring much extra recording overhead.

4. RECORD-REPLAY RUNTIME

After computing a desirable replay interface, as described in Section 3, iTarget instruments the target program accordingly during compilation to insert calls that are linked to its runtime for recording and replay. This section describes iTarget's runtime mechanisms that ensure control and data flow, memory footprints, and thread interleaving do not change from recording to replay.

4.1 Calls, Reads and Writes

When computing a replay interface, iTarget partitions functions (operation nodes) and variables (value nodes) in a static flow graph into replay and non-replay spaces. Since functions in non-replay space will not be executed during replay, iTarget must record all side effects from non-replay space.

First, iTarget records function calls from non-replay space to replay space. Consider a function f in non-replay space that calls function g in replay space. During replay, f will not be executed, and is not able to call g, which should be executed; the iTarget runtime does so instead. Specifically, for each call site in a function placed in non-replay space, iTarget resolves the callee to see whether it *must* belong to non-replay space: if yes, iTarget does not record anything; otherwise iTarget logs the call event during recording, and issues the call during replay when the callee does belong to replay space.

Furthermore, iTarget instruments necessary instructions to record data that flow from non-replay space to replay space. Specifically, iTarget instruments 1) instructions placed in replay space that *read* from variables in non-replay space, and 2) instructions placed in non-replay space that *write* to variables in replay space.

We refer the two kinds of instructions to be instrumented above as "read" and "write", respectively. Other instructions remain unchanged so that they can run at native speed.

When executing a "read" instruction in the original run, the runtime records the values being read in the log. When executing the same instruction during replay, the runtime simply feeds back the values from the log, rather than letting it read from memory.

It is more complex to record and replay a "write" instruction. Since the instruction is never executed during replay, the runtime has to instead issue the write to memory. In addition to the values to be written, the runtime needs to know *where* and *when* to do so.

To determine where to issue the writes, the runtime records the memory addresses that the instruction is writing to, along with the values, so that it can write the values back to the recorded memory addresses during replay.

To determine when to issue the writes, the runtime further orders writes with calls. Consider a function f in non-replay space, which writes to variable x, makes a call to function g that is in replay space, and then writes to variable y. The runtime records the three events of writing x, calling g, and writing y in the original run; it then issues the three events in the *same* order during replay.

4.2 Memory Management

To ensure correctness, addresses of variables in replay space should not change from recording to replay. This is non-trivial because functions in the non-replay space may allocate memory (e.g., by calling `malloc`) and will not be executed during replay; iTarget needs to ensure that memory addresses returned by subsequent calls to `malloc` in replay space are the same as those during recording.

For values allocated on the heap, iTarget uses a separate memory pool for the execution in replay space. Any call to `malloc` in replay space will be redirected to that pool. Since the execution in replay space remains the same from recording to replay, so are the memory addresses allocated in the pool.

For values allocated on the stack, iTarget could run functions in replay and non-replay spaces on two separate stacks, like Jockey [31] and R2 [11]; the runtime would switch the stacks when crossing the two spaces, which introduces additional overhead.

iTarget employs a more lightweight approach. It uses only one stack, and ensures deterministic stack addresses by guaranteeing the value of the stack pointer (ESP on x86) does not change from recording to replay when the program enters a function in replay space. To do so, the iTarget runtime records current ESP value before a call from non-replay space to replay space in the original run. During replay, the runtime sets ESP to the recorded value before issuing the call, and restores ESP and the stack after that. If current ESP is lower than the recorded one, iTarget also backs up the data in the overlapped range to avoid overwriting.

4.3 Thread Management

As we have discussed in Section 2.3, iTarget supports several multithreading strategies. iTarget runs in the mode of causal dependence tracking by default, so the resulting log contains both data flow and synchronization information. For asynchronous signals, iTarget uses the standard technique [10, 11, 22] to delay the delivery until safe points.

5. CHOOSING REPLAY TARGETS

iTarget allows programmers to choose appropriate replay targets and therefore enables *target replay*. This added flexibility can translate into significant savings in recording overhead compared to whole-program replay. This section discusses several possible ways for choosing replay targets, as well as the resulting replay interfaces.

5.1 Modular Programs

Applications that emphasize a *modular* design come with a natural boundary for choosing replay targets. For example, Apache HTTP Server is designed and implemented as a main backbone plus a set of plug-in modules, as shown in Figure 1. The programmer developing a plug-in module neither has to debug the backbone, which is generally stable, nor other modules, which are largely irrelevant. She can simply choose all the source code of her own module as the replay target for recording and replay.

A second example, Berkeley DB, also uses a modular design for building a replicated database. On each node running Berkeley DB, there is a replication manager that coordinates with other nodes, as well as a traditional storage component that manages on-disk data. When debugging the replication protocol, a programmer can choose the code of the replication manager as the replay target, ignoring the mature and irrelevant storage component that may involve massive I/O communications.

Section 6.2 provides case studies for the two applications. The replay interface computed by iTarget often approximates the boundary between modules. The insight is that a modular design implies

that each module mostly uses internal data structures; the rest of the program may not be involved much. A programmer can choose the module of her interest as a replay target; iTarget's minimum cut can exploit the structure manifested in design and computes a replay interface that results in smaller overhead.

5.2 Monolithic Programs

For a monolithic program that does not have a clear component boundary, such as certain algorithm implementations, a programmer can simply choose the entire program as the replay target, which falls back to a whole-program replay. Even in this case, if the program does not directly manipulate all of its input, iTarget will record only the necessary data and skip the payload.

Note that a programmer can still choose a subset of functions as replay target. The risk is that the replay target may be tightly coupled with the rest of the program, exchanging a large amount of data. It could possibly lead to even higher recording overhead to choose the replay interface naively. Fortunately, iTarget can avoid such an anomaly through computing the minimum cut as the replay interface. It is expected that in the worst case iTarget will resort to that of whole-program replay. Section 6.3 uses SPEC CINT2000 benchmarks to illustrate such cases.

5.3 Crash Points

In practice, when a running program crashes at some program point, a programmer may choose code pieces related to that crash point as the replay target. This can be done by simple heuristics, e.g., picking up all functions in the same source file, or by automatic tools, e.g., program slicing [12, 37, 40] or related function investigation [18, 25]. The topic is beyond the scope of this paper.

6. EVALUATION

We have implemented iTarget for C programs on Windows x86. The analysis and instrumentation components are implemented as plug-ins within the Phoenix compiler framework [20].

We have applied iTarget to a variety of benchmarks, including Apache HTTP Server 2.2.4 with service modules, Berkeley DB 4.7.25 with the fault-tolerant replication service, the neon HTTP client 0.28.3, the Wget website crawler 1.11.4, and six programs from SPEC CINT2000. The code sizes of the benchmarks also span a wide range from small (SPEC CINT2000: 5$\sim$10 KLOC), medium (neon & Wget: 10$\sim$50 KLOC), to very large (Apache & Berkeley DB: 100+ KLOC). Their variety and complexity extensively exercise both the static interface analysis and the record-replay runtime of iTarget, leading to a thorough evaluation.

We checked the correctness of replay run by making sure it successfully consumes all the logs generated during recording run. We also manually spot-checked some of the executions through attaching debugger into both recording and replay runs, and comparing their internal states in replay space.

The rest of the section answers the following questions. 1) How effective is target replay in reducing recording overhead? 2) How well does iTarget perform when the replay target is the whole program? 3) Is the effectiveness of iTarget sensitive to the data used in the profiling run? 4) What is the computation cost for finding an efficient replay interface?

6.1 Methodology

We categorize the benchmarks into two sets according to program modularity. One set includes Apache and Berkeley DB, both of which contain natural module boundaries. We apply both target and whole-program replay to them for evaluating the effectiveness of target replay. To reflect the fact that a programmer typically works on individual modules, we replay a single module each time during target replay experiments.

The other set consists of network clients and SPEC CINT2000 programs. These programs are implemented in a monolithic way with no natural module boundary. We evaluate the performance of iTarget in the worst case by applying whole-program replay on these benchmarks. We also evaluate the sensitivity of iTarget's effectiveness on profiling runs with different workload scales and present a quantitative computation cost of iTarget.

We additionally run two other recent replay tools, iDNA [2] and R2 [11], on all benchmarks for comparison. iDNA is built on top of an instruction-level simulator and inspects each instruction for recording and replay, while R2 interposes at function level and requires developers to manually annotate the replay interface to decide what should be recorded. In our experiments, iDNA imposes more than five times slowdown and generates significantly larger logs than iTarget and R2 for all benchmarks. It even fails to terminate when recording `mcf` and `vpr` after producing 50 GB logs. Therefore we omit its data on the detailed discussion of each benchmark, and only compare the overhead of iTarget (log size and slowdown) with those imposed by R2.

Our experiments on two server applications, Apache and Berkeley DB, were conducted on machines with 2.0 GHz Intel Xeon 8-way CPU and 32 GB memory. Other experiments were conducted on machines with 2.0 GHz Intel Xeon dual-core CPU and 4 GB memory. All of these machines are connected with 1 Gb Ethernet, running Windows Server 2003. In all experiments, iTarget shares the same disk with the application.

6.2 Performance on Modular Programs

We use Apache HTTP Server and Berkeley DB to evaluate the performance advantage of target replay on modular programs. We further investigate how iTarget computes an appropriate replay interface to isolate the target module with only slight recording overhead. Our experiments include replaying both whole programs and individual modules. Table 1 lists the modules used for target replay.

Note that R2 cannot replay individual modules of these programs, so we only present data of whole-program replay. In fact, although R2 supports the replay of part of a program, it requires the programmer to annotate the side effects of the functions that compose the replay interface. However, its annotation language only supports to specify plain buffers [11], rather than side effects that involve complex pointer usages, which pervade Apache and Berkeley DB functions. Besides, it would be tedious and error-prone to manually annotate hundreds of functions.

Apache HTTP Server. Apache comes with a flexible modular framework. The core part `libhttpd` invokes responsible modules to handle HTTP requests. We include three commonly-used modules in our experiment, namely `mod_alias`, `mod_dir`, and `mod_deflate`, listed in Table 1. Specifically, `mod_alias` maps request URLs to filesystem paths; `mod_dir` provides "trailing slash" redirection for serving directory index files; and `mod_deflate`, acting as an output filter, compresses files before sending them to clients.

In the experiment, Apache HTTP Server runs with all these three modules; we use iTarget to replay each of them individually. Since each module contains a single source file, to replay a module we choose all functions in the corresponding file as the replay target.

We set up the built-in Apache benchmarking tool `ab`, which starts clients that repetitively fetch files from a server via HTTP requests. We put an HTML file `index.html` sized 256 KB on the server and start eight clients to grab the compressed version of the file via an aliasing name of the directory. Thus for each request, all

	Replay Target	Replay Target Sources	Description
Apache	`mod_alias`	mod_alias.c	Mapping URLs and file paths
	`mod_dir`	mod_dir.c	Serving directory index files
	`mod_deflate`	mod_deflate.c	Compressing HTTP output
Berkeley DB	`repmgr`	All 12 files in the "repmgr" directory	Replication service

Table 1: Modules used for target replay.

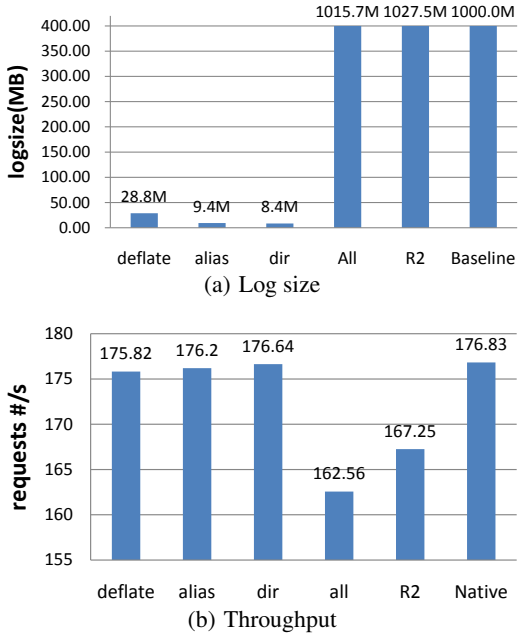


Figure 5: Recording performance of different replay targets in Apache.

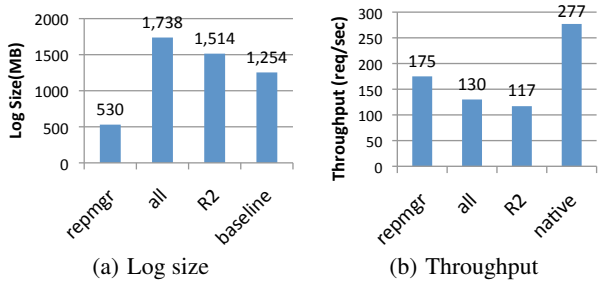


Figure 6: Log size and throughput of recording Berkeley DB.

three modules are executed. iTarget uses 40 requests for profiling in order to assign costs to edges in the static flow graph; clients send 4,000 requests in the experiment.

Figure 5(a) shows log sizes generated for answering client requests when iTarget is replaying each module and the whole Apache program respectively, where the baseline is the total size of data that Apache reads from disk and network. The log sizes remain small as iTarget tries to replay only an individual module. iTarget consumes less than 10 MB log when replaying `mod_alias` and `mod_dir`. Replaying the more complex module `mod_deflate` takes 29 MB, which is still substantially smaller than the baseline. This shows that iTarget manages to avoid logging the entire file and network I/O, and only record the necessary data for replaying a single module. For example, to correctly replay module `mod_deflate`, iTarget only needs to record the metadata exchanges and skips the entire file content, which is manipulated in the underlying third-party library `zlib`. On the contrary, the log sizes of both whole-program replay and R2 are close to the baseline (1 GB).

Figure 5(b) shows the throughput during recording for each replay target. The throughput decreases as the log size increases. Replaying a single module using iTarget inflicts only less than 1% performance slowdown. However, a whole-program replay incurs 8% slowdown, though the performance is still comparable to R2.

Berkeley DB. In the experiment on Berkeley DB, we start two nodes to form a replication group. One node is elected as the master, and randomly inserts 5,000 key-value pairs (sized 2 KB each) to the replicated database. We use iTarget to replay one node.

We choose to replay the `repmgr` module that implements a distributed replication protocol. It is known to have subtle bugs and hard to debug [39]. We specify all functions in 12 source files of this module as the replay target.

Figure 6(a) shows the log sizes of iTarget and R2 with different interfaces. It also shows the baseline log size as the size of all input data from disk and network. Note that the baseline volume to be recorded is much larger than the application's input data, because when each node receives log records from the master, it may scan its own database log file to find the corresponding records belonging to the same transaction, which incurs substantial file reads.

The result shows that the log size of iTarget for replaying the `repmgr` module is only about $1/3$ of that of iTarget for whole-program replay and that of R2. This is because `repmgr` does not directly touch all the data read from the database log file. iTarget only needs to record values returned by lower file I/O and local database modules, thereby substantially reducing the log size.

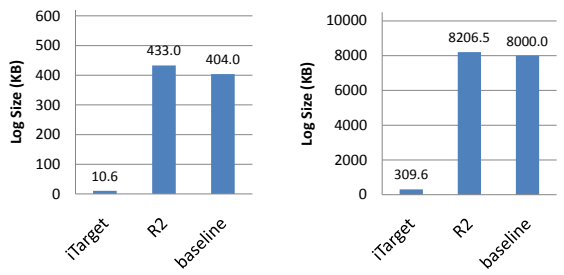
Both the log sizes of iTarget for whole-program replay and R2 are larger than the baseline. This is due to the cost of tracking the causal order of synchronization events in the synchronization log. It turns out that Berkeley DB heavily uses the interlocked operations, leading to the excessive cost.

Figure 6(b) shows Berkeley DB's throughput during recording with iTarget for different replay targets and with R2. iTarget and R2 incur 53% and 58% slowdown in whole-program replay, respectively. This is mostly due to the cost of tracking interlock operations. However, when using target replay on the `repmgr` component, iTarget incurs only 37% slowdown, thus achieving 35% and 50% throughput improvements compared to the previous two whole-program replay cases, respectively.

The above experiments demonstrate that iTarget can automatically identify a correct replay interface that separates a single module from the surrounding environment. For modular programs like Apache HTTP Server and Berkeley DB, iTarget enables significant performance improvements through target replay of modules.

6.3 Performance on Monolithic Programs

We evaluate iTarget's recording performance of whole-program replay on monolithic programs. For monolithic programs that do not directly manipulate input data, e.g., an HTTP client that parses



(a) Log size for neon (b) Log size for Wget

Figure 7: Log size for recording neon and Wget.

the HTTP header and ignores the content of the HTTP body, iTarget is able to exploit this to reduce recording overhead; we use two HTTP clients, neon and Wget, to illustrate this case. For monolithic programs that implement certain algorithms and perform intensive computations, their executions depend heavily on every piece of input data and replay tools need to record all input; we use six SPEC CINT2000 programs to illustrate that iTarget's performance is still comparable to existing replay tools in this case.

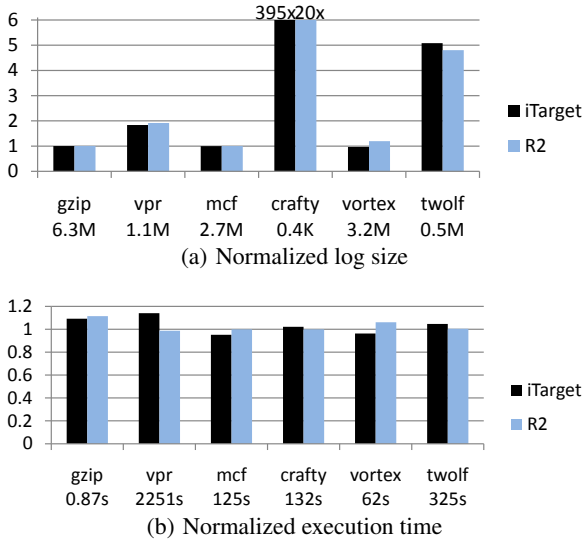
The programmer can specify a subset of functions as the replay target for a monolithic program. Even if the replay target may exchange a large amount of data with the rest of the program, iTarget can automatically detect the issue and avoid recording the data by finding a lower-cost replay interface elsewhere. It guarantees that the resulting performance is no worse than that of the whole-program replay. We use `crafty`, one of the SPEC CINT2000 programs, to illustrate the case.

Network clients. We use HTTP clients neon and Wget as benchmarks to evaluate iTarget's performance. For HTTP clients, the most essential part is to handle the HTTP protocol, which is only related to the header of input data. In all our experiments for neon and Wget, the slowdown caused by replay tools is negligible. This is because the performance bottleneck is mainly the network or the server side. We therefore present only the log sizes here.

The neon client skips the payload in HTTP responses. We set up an HTTP server and run neon on another machine to request files repetitively; the average size of those files is 400 KB. Figure 7(a) shows the log sizes for neon. The size of the data downloaded from the network is the baseline for replay. iTarget successfully avoids recording the payload data, reducing logs to around 2.5% of the original. iTarget records only the HTTP headers that neon inspects, while R2 records the entire HTML files.

Wget is a web crawler that crawls a web site and converts its pages for offline browsing. We set up a mini web site and make Wget crawl its pages. Each HTML file is 400 KB and the total size of the crawled HTML files is 4 MB. Figure 7(b) shows the log sizes for Wget. Unlike neon, Wget parses each downloaded file to find new hyperlinks. The baseline log size is twice of HTML files size due to extra disk I/O during parsing. iTarget still shows its advantage, because Wget touches the payload only via libc functions such as `stricmp`. iTarget then only records their return values to avoid recording the whole file data. It reduces the logs to only 309.6 KB.

The above experiments show that, with the help of the language-based model, iTarget can identify the payload in an input file and skip it. Thus, for the applications that do not directly manipulate all of their inputs, such as neon and Wget, iTarget outperforms previous tools even with whole-program replay.



(a) Normalized log size

(b) Normalized execution time

Figure 8: Recording cost of whole-program replay for SPEC CINT2000 programs.

SPEC CINT2000. We evaluate the performance on six SPEC CINT2000 applications using standard test suites. Although many of them are actually deterministic given the same input, they are good for evaluating the worst cases of iTarget, where neither target replay nor skipping payload is possible. We use iTarget to replay the whole program of each benchmark and compare the slowdown and the log size of iTarget with R2.

Figure 8(a) shows the log sizes, normalized to input data size (shown under each label). It is not surprising that iTarget requires to record as much data as the input file. `vpr` and `twolf` read files multiple times, thereby causing the large log sizes of iTarget and R2. The input data size of `crafty` is less than 500 bytes, and the log of both iTarget and R2 is dominated by auxiliary tags and events (e.g., calls). This explains the large log size ratio of `crafty`.

Figure 8(b) shows the slowdown, normalized to the native execution time (shown under each label). We can see that iTarget has similar performance to R2 and native in all SPEC CINT2000 benchmarks. iTarget and R2 may run faster than native execution sometimes, because they redirect `malloc` and `free` in replay space to a separate memory pool with full-fledged memory management functionality, which has slightly different performance from the libc counterpart (see Section 4.2).

The result shows that, for CPU-intensive benchmarks, both the log size and slowdown of iTarget to replay whole programs are comparable to those of R2.

Target replay in crafty. In SPEC CINT2000 programs, functions have heavy data dependencies. Only replaying a single function can even result in a worse overhead than whole-program replay, if done naively. iTarget will automatically choose an appropriate replay interface, while a naive cut mentioned in Section 2.2 may generate a huge log, especially when the replay target repeatedly reads data from other parts of the program, as seen in some algorithm implementations.

We use `crafty` as a test case, where we choose `SearchRoot`, a function that implements the alpha-beta search, as the replay target. iTarget detects the heavy dependencies between this function and the rest part of `crafty`; the resulting replay interface leads to a log similar to that of a whole-program replay in size (both 159 KB). In contrast, a naive cut strategy produces a huge log (17 GB).

	# of Req.	Size per Req. (KB)	Log Size (MB)
Apache	None	None	1042.61
(`mod_deflate`)	40	256	28.80
	4,000	25	29.15
	4,000	256	28.80
Berkeley DB	None	None	1,765.27
(`repmgr`)	50	2	530.07
	5,000	0.2	643.96
	5,000	2	531.42

Table 2: The log size of Apache and Berkeley DB under different profiling settings.

Prog.	KLOC	Op.	Val.	Edges	Cut Time (s)
mcf	2	57	6,796	10,446	0
gzip	8	156	20,160	17,722	1
vpr	17	311	54,274	75,476	1
neon	19	481	38,960	163,750	2
twolf	20	219	103,824	719,707	3
crafty	21	153	78,630	80,548	2
wget	41	623	85,946	457,953	15
vortex	67	982	204,342	545,863	2
apache	141	2,362	350,906	34,349,476	202
bdb	172	2,048	746,970	82,292,406	300

Table 3: Statistics of computation cost of replay interfaces. "Op.", "Val.", and "Edges" list the numbers of operation nodes, value nodes, and edges in static flow graphs, respectively; "Cut Time" lists the time for computing the minimum cut; "bdb" is short for Berkeley DB.

The experiments show that iTarget is capable of intelligently skipping the unused payload within input during whole-program replay of monolithic programs, therefore reducing log sizes significantly. In the worst case, where programs depend on all input data and the replay target is tightly coupled with the rest of the program, target replay in iTarget will automatically fall back to whole-program replay. Even in those cases, the overhead of iTarget remains comparable to that of existing tools.

6.4 Profiling Workload and Computation Cost

Profiling workload. We evaluate how different profiling workload scales could affect the resulting replay interfaces. Table 2 shows the log sizes for replaying the module `mod_deflate` in Apache HTTP Server and the module `repmgr` in Berkeley DB under different profiling workload scales. "None" represents the recording log size without any profiling run. In this case, we simply assign the same weight to each edge in static flow graph.

The result shows that the profiling run is important for reducing overhead of iTarget. Without profiling, the log sizes of Apache and Berkeley DB are 35 times and twice larger, respectively. However, iTarget is not sensitive to different workload scales of a profiling run. In Apache and Berkeley DB, the resulting log sizes do not change much when using different input file sizes or different numbers of requests for profiling.

Computation cost. Table 3 shows the statistics of computation cost of replay interfaces. For each program, we report the numbers of operation nodes, value nodes, and edges in the static flow graph, as well as the time it takes to compute the minimum cut. The pointer analysis can finish within seconds for all the benchmarks so we omit the time.

In general, iTarget is efficient to find a near-optimal replay interface for a target function set. If a programmer chooses different targets or modifies the code, iTarget can reconstruct the graph and recompute the replay interface within minutes.

6.5 Summary

In terms of performance, iTarget enjoys two advantages: imposing small instrumentation overhead and reducing logging I/O that competes with application I/O. This makes iTarget more lightweight when applied to both CPU- and I/O-intensive applications. It also shows that, even in the worst case of whole-program replay, with the minimum cut strategy, iTarget still achieves comparable performance to state-of-the-art replay tools.

The accuracy of profiling is critical for iTarget, but iTarget does not strictly require the same input during profiling and recording. The user does not always need to re-profile the graph if she just increases the testing workload (e.g., in terms of file sizes or numbers of requests) in recording phase.

7. RELATED WORK

Process-level replay. Library-based replay tools like Jockey [31], liblog [10], RecPlay [26], and Flashback [33] adopt a fixed replay interface at library functions and tend to record and replay an entire program. R2 [11] is more related to iTarget since it also employs language-based techniques and is able to replay only a part of a program. R2 asks programmers to select a set of functions as a replay interface to isolate replay targets, and annotate their side effects with a set of keywords. However, it is tedious and error-prone for programmers to select *manually* a *complete* interface that isolates all non-determinism. Furthermore, the expressiveness of R2 keywords is limited; it is difficult to annotate functions that access memory other than plain buffers. iTarget automatically computes an efficient replay interface from source code via program analysis to minimize recording overhead. It interposes itself at the instruction level and requires no programmer annotations.

iDNA [2] also takes an instruction-level replay interface. It uses an instruction-level simulator to track each instruction and to record necessary data for replay, such as register states after certain special non-deterministic instructions and memory values that are read by instructions during execution. To avoid recording all memory values read, iDNA maintains a shadow memory internally to cache previous values. Despite the optimization, iDNA still incurs significant performance slowdown due to expensive instruction tracking. iTarget needs to track only memory access instructions at the computed replay interface and hence incurs remarkably less overhead.

There are a number of replay tools focusing on applications using various programming language runtimes, such as Java [16, 24, 34], MPI [27], and Standard ML [35]. While current iTarget implementation works with the C language, its language-based technique and data flow model are general enough and can be easily applied to other programming languages.

Whole-system replay. Hardware based [21, 38] and virtual machine based [7, 15] replay tools aim to replay a *whole* system, including both target applications and the underlying operating system. Because these tools intercept a system at a low-level interface, such as at the processor level, it is easy for them to observe all the non-determinism from the environment. Special hardware or virtual-machine environment is required for those tools to work.

Language-based security. Swift [3] provides a secure programming language to construct secure web applications and partitions a control flow graph to split programs into a client and a server; iTarget partitions a program into replay and non-replay spaces through a cut on data flow graph to ensure determinism for replay target. FlowCheck [19] is a tool quantitatively estimating leak of secure data by dynamically tracking information flows; iTarget estimates its flow graph statically.

8. CONCLUSION

The beauty of iTarget lies in its simple and general model that defines the notion of correct replay precisely. The model leads to our insight that the problem of finding an optimal replay interface can be reduced to that of finding the minimum cut in a data flow graph. With this model, iTarget employs programming language techniques to achieve both correctness and low recording overhead when replaying complex, real-world programs.

Acknowledgments

We would like to thank Ivan Beschastnikh, Frans Kaashoek, Lintao Zhang, and the anonymous reviewers for their valuable feedback on early drafts of this paper.

9. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization of the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[3] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *SOSP*, 2007.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[5] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[6] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[7] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[8] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE*, 2006.

[9] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, 2007.

[10] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX ATC*, 2006.

[11] 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 *OSDI*, 2008.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.

[13] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, 2005.

[14] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[15] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC*, 2005.

[16] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *IPDPS*, 2000.

[17] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging deployed distributed systems. In *NSDI*, 2008.

[18] F. Long, X. Wang, and Y. Cai. API hyperlinking via structural overlap. In *ESEC/FSE*, 2009.

[19] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, 2008.

[20] Microsoft. The Phoenix compiler framework. `http://research.microsoft.com/phoenix/`.

[21] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[22] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[23] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[24] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *WODA*, 2005.

[25] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE*, 2005.

[26] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *TOCS*, 17(2):133–152, 1999.

[27] M. Ronsse, K. D. Bosschere, and J. C. de Kergommeaux. Execution replay for an MPI-based multi-threaded runtime system. In *ParCo*, 1999.

[28] M. Ronsse, M. Christiaens, and K. D. Bosschere. Cyclic debugging using execution replay. In *International Conference on Computational Science*, 2001.

[29] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL*, 1998.

[30] D. Saff and M. D. Ernst. Mock object creation for test factoring. In *PASTE*, 2004.

[31] Y. Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG*, 2005.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *TOCS*, 15(4):391–411, 1997.

[33] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *USENIX ATC*, 2004.

[34] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA*, 2000.

[35] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[36] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *ESEC/FSE*, 2007.

[37] M. Weiser. Program slicing. In *ICSE*, 1981.

[38] M. Xu, R. Bodík, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[39] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI*, 2009.

[40] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE*, 2006.