DrDebug: Deterministic Replay based Cyclic Debugging with Dynamic Slicing

Yan Wang*, Harish Patil**, Cristiano Pereira**, Gregory Lueck**, Rajiv Gupta*, and Iulian Neamtiu*

*CSE Department, UC Riverside , {wangy,gupta,neamtiu}@cs.ucr.edu

**Intel Corporation , {harish.patil,cristiano.l.pereira, gregory.m.lueck}@intel.com

ABSTRACT

We present a collection of tools, DrDebug, that greatly advances the state-of-the-art of cyclic, interactive debugging of multi-threaded programs based upon the record and replay paradigm. The features of DrDebug significantly increase the efficiency of debugging by tailoring the scope of replay to a buggy *execution region* or an *execution slice* of a buggy region. In addition to supporting traditional debugger commands, DrDebug provides commands for recording, replaying, and *dynamic slicing* with several novel features. First, upon a user's request, a highly precise dynamic slice is computed that can then be browsed by the user by navigating the dynamic dependence graph with the assistance of our graphical user interface. Second, a dynamic slice of interest to the user can be used to compute an execution slice whose replay can then be carried out. Due to narrow scope, the replay can be performed efficiently as execution of code segments that do not belong to the execution slice is skipped. We also provide the capability of allowing the user to step from the execution of one statement in the slice to the next while examining the values of variables. To the best of our knowledge, this capability cannot be found in any other slicing tool. We have also integrated DrDebug with the Maple tool that exposes bugs and records buggy executions for replay. Our experiments demonstrate DrDebug's practicality.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

design, experimentation

Keywords

deterministic replay, execution slice

CGO '14, February 15 - 19 2014, Orlando, FL, USA

ACM 978-1-4503-2670-4/14/02 $15.00.

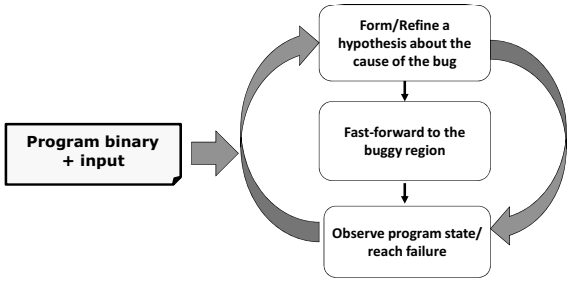


Figure 1: Cyclic Debugging Process

1. INTRODUCTION

Cyclic debugging is the iterative process of narrowing down the reason for a program failure. The failure, or the bug, has a root cause (where the bug is introduced) and a symptom (where the bug's effect is seen). In each debug iteration, the programmer gathers more information, adding to the knowledge gained from previous debug iterations, finally zeroing in on the root cause of the bug. The process, outlined in Figure 1, involves making a hypothesis about the cause of the bug, fast-forwarding to the buggy region, and performing a detailed state examination until the bug manifests itself. The cycle is repeated until the root cause of the bug is found.

Cyclic debugging poses multiple challenges:

1. Depending on the location of the bug, it can take a very long time to fast-forward and reach it.
2. Many aspects of the program state, such as heap/stack location, outcome of system calls, thread schedule, change between debug sessions.
3. Some bugs are hard to reproduce, in general and also under a debugger.

To address these challenges we introduce a Deterministic replay based Debugging framework, or DrDebug for short. It is a collection of tools based on the program capture and replay framework called PinPlay [23]. PinPlay uses the Pin dynamic instrumentation system [18]. PinPlay consists of two pintools: (i) a *logger* that captures the initial architecture state and non-deterministic events during a program execution in a set of files collectively called a *pinball*; and (ii) a *replayer* that runs on a pinball repeating the captured program execution therein.

Our proposed PinPlay-based cyclic debugging process is outlined in Figure 2. It involves two phases: (i) capturing the buggy region in a pinball using the PinPlay logger; and (ii) replaying the pinball and use Pin's advanced debugging

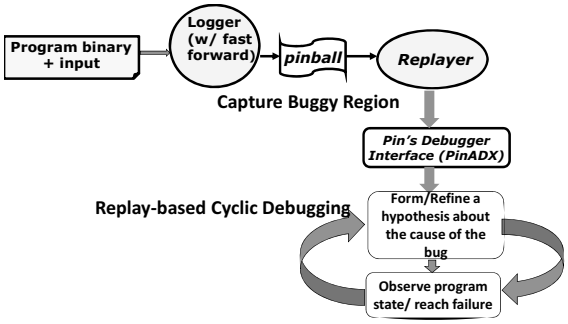


Figure 2: Cyclic Debugging with DrDebug

extension *PinADX* [17] to do cyclic debugging. Our process addresses various debugging challenges as follows:

1. The programmer uses the logger to fast-forward to the buggy region and then starts logging until the bug appears. Thus the generated pinball captures only an *execution region* that includes both the root-cause and the symptom of the bug. During replay-based debugging, each session starts right at the entry of the buggy region avoiding the need for fast-forwarding.
2. The programmer observes the exact same program state (heap/stack location, outcome of system calls, thread schedule, etc.) during multiple debug sessions based on the replay of the same pinball.
3. If the logger manages to capture a buggy pinball, it is guaranteed that the bug will be reproduced on each iteration of cyclic debugging. For hard-to-reproduce bugs, the logger can be combined with bug-exposing tools such as Maple [30] to expose and record the bug.

PinPlay also enables deterministic analysis of multi-threaded programs via pintools built for analysis during replay. PinADX can make the analysis available to the user as a set of extended debugger commands. Using these two capabilities, we have designed a practical (efficient and highly precise) dynamic slicer for multi-threaded programs. The dynamic slice of a computed value identifies all executed statements that directly or indirectly influence the computation of the value via dynamic data and control dependences [14]. In this paper we greatly advance the practicality of dynamic slicing by (i) slicing execution regions to control the high cost of slicing, (ii) making a slice available across multiple debug sessions, (iii) allowing forward navigation of a slice in a live debugging session, (iv) improving its precision, and (v) handling multi-threaded programs. The result is a replay debugging tool, consisting of gdb with a KDbg graphical user interface, that allows users to interactively query the statements affecting a variable value at a specific statement. Slices found once are usable across multiple debug sessions because of PinPlay's repeatability guarantee.

The key contributions of this work are as follows:

1. A working debugger (gdb) with a graphical user interface (KDbg) that allows debugging based on replay of pinballs for multi-threaded programs. All regular debugging commands (except state modification) continue to work. In addition, new commands for region recording and dynamic slicing are made available.
2. Handling of dynamic program slicing for multi-threaded programs. The slicing works for a recorded region from a program execution. We have implemented new optimizations to make interactive slicing practical and developed analysis to make the computed slices *highly precise*.
3. Leveraging PinPlay's capabilities, we have developed a logging tool for capturing an *execution slice* which allows us to replay the execution of statements included in a dynamic slice efficiently by skipping the execution of code regions that do not belong to the slice. Programmers can load a previously generated slice and step forward from the execution of one statement in the slice to the next while examining values of program variables at each point. Such support is not provided in any prior dynamic slicing tool as they merely permit examination of slice after program execution.
4. We modified the Maple tool-chain [4] for recording the buggy executions it exposes. The resulting pinball can be readily used by DrDebug.

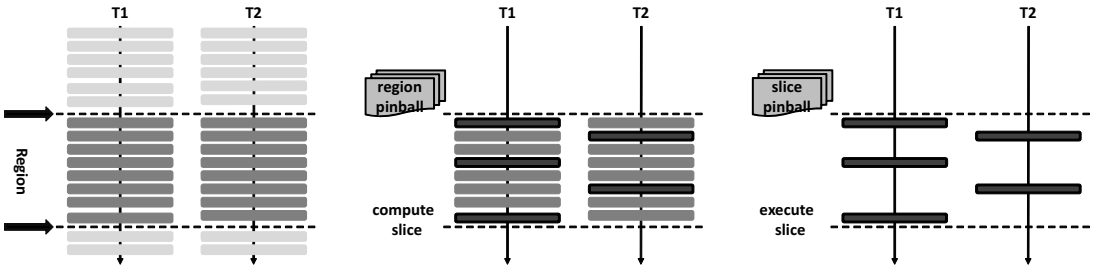
Both PinPlay and dynamic slicing can incur a large runtime overhead. Luckily, for cyclic debugging their use can be restricted only to the buggy region. The overhead actually seen by the users will depend on the lengths of their buggy region. In a study of 13 buggy open source programs [21] the buggy region length (called *Window size* in the paper) was typically less than 10 million instructions, and at most 18 million instructions. In our experiment with eight 4-threaded PARSEC [10] program runs, on average, regions with 100 million instructions in the main thread (541 million instructions in all threads) could be logged in 29 seconds and replayed in 27 seconds. We also found the overhead for region-based slicing to be quite reasonable – a few seconds to a few minutes for regions of average length 6 million instructions.

The remainder of this paper is organized as follows. Section 2 provides an overview of DrDebug features. Section 3 presents our replay-based dynamic slicing algorithm for multi-threaded programs. Section 4 discusses execution slicing and Section 5 presents techniques for improving the precision of dynamic slicing. Section 6 gives an overview of the implementation and Section 7 presents our evaluation. Related work and conclusions are given in Sections 8 and 9.

2. OVERVIEW OF DRDEBUG

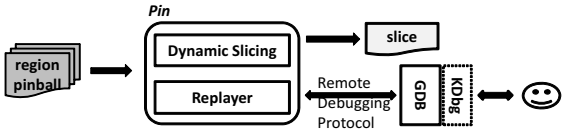
Debugging begins once a *pinball* that captures a failing run is available. This initial pinball is either generated automatically, using a testing tool, or with the assistance of the programmer. In the former case we use the Maple bug exposing tool then capture the corresponding pinball. In the latter case, we provide GDB commands/GUI buttons so the programmer can fast-forward to the buggy region and then manually capture the pinball. DrDebug is designed to achieve two objectives: *replay efficiency* - so that it can be used in practice; and *location efficiency* - so that the user's effort in locating the bug can be reduced.

Replay efficiency. In designing DrDebug, one of our key objectives is to speedup debugging by improving the speed of replay. This is particularly important for long program executions. We tackle this problem by narrowing the scope of execution that is captured by the *pinball* using the notions of *Execution Region* and *Execution Slice* in DrDebug.

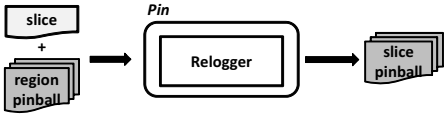


(a) Original Execution (b) Replaying *Execution Region* (c) Replaying *Execution Slice*

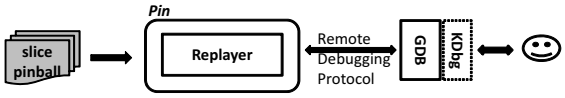
Figure 3: Narrowing Scope of Execution for Replay.



(a) Replay Execution Region; Compute Dynamic Slices.



(b) Generate Slice Pinball from Region Pinball.



(c) Replay Execution Slice and Debug by Examining State.

Figure 4: Dynamic Slicing in DrDebug

- *Execution Region* - instead of collecting the pinball for an entire execution, users can focus on a (buggy) region of execution by specifying its start and end points. The *region pinball* is then drives replay-based debugging.
- *Execution Slice* - when studying the program behavior along a dynamic slice, instead of replaying the entire execution region, we exclude the execution of parts of the region that are not related to the slice. This is enabled by replaying using the *region pinball* and performing relogging to collect the *slice pinball*.

Figure 3 shows how the scope of execution that is replayed is narrowed down by the above two techniques. This greatly increases the speed of replay and makes replay based debugging practical for real world applications.

Location efficiency. To assist in locating the root cause of failure, we provide the user with a *dynamic slicing* capability. The components of the dynamic slices and their usage are (see Figure 4):

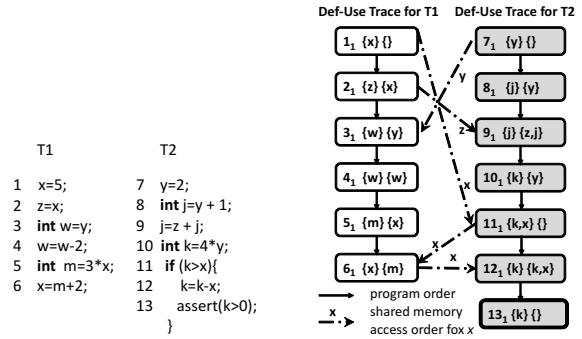
- When the execution of a program is replayed using the *region pinball*, our slicing pintool collects dynamic information that enables the computation of dynamic slices. Requests for dynamic slices are made by the programmer and the computed slices can be browsed or traversed going backwards along the dynamic dependences using our KDbg-based graphical user interface (see Figure 4(a)).
- A dynamic slice of interest found in the preceding step can be saved by the user. This slice essentially identifies a series of points in the program's execution at which the user wishes to examine the program state in greater detail. To prepare for this examination, we generate the *slice pinball* that only replays the execution of statements belonging to the slice. The *relogger* is responsible for generating the *slice pinball* from the computed *slice* by replaying using the *region pinball* (see Figure 4(b)).
- Finally, the user can replay the execution slice using the *slice pinball*. During this execution, breakpoints are automatically introduced allowing the user to step from the execution of one statement in the slice to the next. At each of these points, the user can examine the program state to understand program behavior (see Figure 4(c)).

In contrast to prior work on dynamic slicing, we make the following contributions. First, we develop a dynamic slicing algorithm that not only handles multi-threaded programs, but is integrated with the replay system. Second, we provide a graphical interface which allows the user to browse a dynamic slice by traversing it backwards and examine the program state along the dynamic slice by single stepping-forward as the program executes. Finally, we have developed extensions for capturing dynamic data and control dependences that make the dynamic slice more precise. Next, we present each of these contributions in greater detail.

3. COMPUTING DYNAMIC SLICES

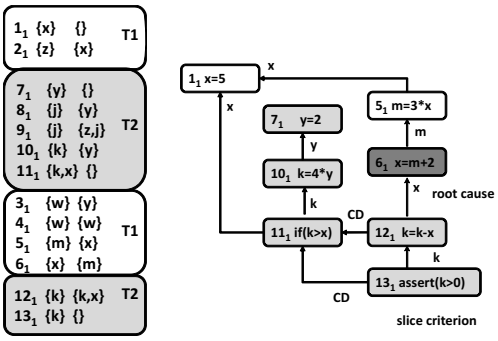
The dynamic slice of a computed value is defined to include the executed statements that played a role in the computation of the value. It is computed by taking the transitive closure over data and control dependences starting from the computed value and going backwards over the dynamic dependence graph. As the execution of a multi-threaded program is being replayed, the user can request the computation of a dynamic slice for a computed value at any statement via our debugging interface. The slice is computed as follows:

(i) Collect Per Thread Local Execution Traces. During replay, for each thread, we collect its local execution trace that includes the memory addresses and registers defined (written) and used (read) by each instruction. This information is needed to identify dynamic dependences.



(a) Example Code.

(b) Per Thread Traces and Shared Memory Access Order.



(c) Global Trace

(d) Slice for k at 13_1.

Figure 5: Dynamic Slicing a multi-threaded program.

(ii) Construct the Combined Global Trace. Prior to slice computation, we combine all per-thread traces into a single fully ordered trace such that each instruction in the trace honors its dynamic data dependences including all read-after-write, write-after-write, and write-after-read dependences. The construction of this global trace requires the knowledge of shared memory access ordering to guarantee that inter-thread data dependences are also honored by the global trace. This information is already available in a pinball, as it is needed for replay. The combined global trace is thus based on the topological order of the graph, in which each execution instance is represented as a node, and an edge from node n to m means that n happens before m either in program order or in shared memory access order.

(iii) Compute Dynamic Slice by Backwards Traversing the Global Trace. A backward traversal of the global trace is carried out to recover the dynamic dependences that form the dynamic slice. We adopted the Limited Preprocessing (LP) algorithm proposed by Zhang et al. [33] to speed up the traversal of the trace. This algorithm divides the trace into blocks and by maintaining summary of downward exposed values, it allows skipping of irrelevant blocks.

Next we illustrate our algorithm on the program in Figure 5. The code snippet is shown in Figure 5(a), where two threads, $T1$ and $T2$, operate on three shared variables (x, y, z). The code region (from line 11 to line 13) is wrongly assumed to be executed atomically in $T2$ by the programmer. However, because of the data race between statements at line 6 and line 12, x is modified unexpectedly in $T1$ by statement at line 6, causing the assertion to fail at line 13 in $T2$ (see Figure 5(b)). To help figure out why the assertion failed, the programmer can compute the backwards dynamic slice for k at line 13 in thread $T2$.

Figure 5(b) shows the individual trace for each thread. We collect the def-use information, i.e., the variables (memory locations and registers) defined and used, for each instruction. For example, 12_1 defines k by using k (defined at 10_1) and x (defined at 6_1). In addition to the per thread local traces and the shared memory access ordering used to compute the slice are also shown in Figure 5(b). The shared memory access orders are shown by the inter-thread dashed edges – for example, edge from 6_1 to 12_1 means the write of x at 6_1 in $T1$ happens before the read of x at 12_1 in $T2$. The intra-thread program orders are shown by solid edges – for example, edge $11_1 \rightarrow 12_1$ means that 11_1 happens before 12_1 in $T2$ by program order. The combined global trace for all threads shown in Figure 5(c) is a topological order of all the traces in Figure 5(b).

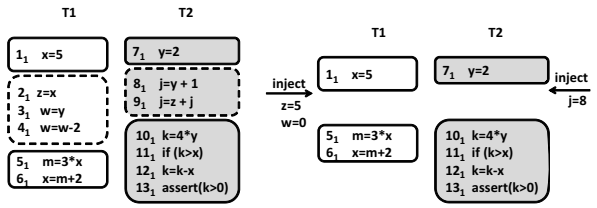
Using the global trace, we can then compute a backwards dynamic slice for the multi-threaded program execution via a backwards traversal of the global trace to recover dependences which should be included in slice. When we construct the global trace in step 2, we always try to cluster traces for each thread to the extent possible to improve the locality of LP algorithm (e.g., after considering 1_1, we continue to consider 2_1 and stop at 3_1 because of the incoming edge from 7_1 to 3_1). The slice for k at 13_1 is shown in Figure 5(d). As we can see, the dynamic slice captures exactly the root cause of the concurrency bug: x is unexpectedly modified at 6_1 in $T1$ when $T2$ is executing an atomic region (assumed by the programmer).

Once a dynamic slice has been computed, the user can examine and navigate the slice using our graphical user interface. In addition, when an interesting slice has been found, the user may wish to engage in deeper examination of how the program state is effected by the execution of statements included in the slice as program execution proceeds. For this purpose, the user can save the slice and take advantage of replaying the execution slice as described in the next section.

4. REPLAYING EXECUTION SLICES

In prior works, the dynamic slice is essentially used for postmortem analysis after program execution. It identifies statement executions that influence the computation of a suspicious value via control and data dependences. To further understand the program behavior; however, the programmer may wish to examine the concrete values of variables at statement instances in the slice to see how these statements impact program state. Therefore we support the idea of *replaying an execution slice* which provides two key features.

- First, the user can examine the values computed along the slice in a live debugging session. In fact we allow the user to step the execution of the program from one statement in the slice to the next statement in the slice.
- Second, for *efficiency*, only the part of computation that forms the slice is replayed. To implement this feature we leverage PinPlay's relogging and code exclusion features.



(a) Code Exclusion Regions. (b) Injecting Values During Replay.

Figure 6: An example of *execution slice*.

PinPlay's relogger can run off a pinball and then generate a new pinball by excluding some code regions. Given a slice, DrDebug can exclude all the code regions which are not in the slice and generate a smaller *slice pinball*. PinPlay's relogger maintains a per-thread exclusion flag to support local exclusion regions for each thread. Given an exclusion code region [$startPc : sinstance : tid, endPc : einstance : tid$) for thread tid, relogger sets the exclusion flag and turns on the side-effects detection when $sinstance^{th}$ execution of $startPc$ is encountered, and then resets the flag when the $einstance^{th}$ execution of $endPc$ is reached in thread tid.

To enable generation of the *slice pinball*, we output a special slice file which, in addition to the normal slice file, also identifies the exclusion code regions. As shown in Figure 6(a), we identify all the exclusion code regions (shown as dashed boxes) for each thread, and output such information to the special slice file. The relogger leverages this file to generate the *slice pinball*. Relogger detects the side-effects of excluded code regions using the same algorithm PinPlay adopted for system call side-effects detection [20]. When DrDebug runs off the *slice pinball*, all the excluded code regions will be completely skipped and their side-effects are restored by injecting modified memory cells and registers as shown in Figure 6(b).

5. IMPROVING DYNAMIC DEPENDENCE PRECISION

The utility of a dynamic slice depends upon the precision with which dynamic dependences are computed. We observe that prior dynamic dependence detection algorithms (e.g., [28, 27, 33, 32]) which leverage binary instrumentation frameworks (e.g., Pin [18], Valgrind [22]) have two sources of imprecision. First, in the presence of *indirect jumps*, these algorithms fail to detect certain dynamic control dependences causing statements to be missed from the dynamic slice. Second, due to the presence of *save* and *restore* operation pairs at function entry and exit points, spurious data dependences are detected causing the dynamic slices to be unnecessarily large. Next we address these sources of imprecision. To the best of our knowledge, we are the first to observe and propose solutions to mitigate these problems.

5.1 Dynamic Control Dependence Precision

For accurately capturing dynamic control dependence in the presence of recursive functions and irregular control flow, we use the online algorithm by Xin and Zhang [27]. However, using this algorithm in the context of a dynamic binary instrumentation framework poses a major challenge. It assumes the availability of precomputed static immediate post-dominator information for each basic block. Due the presence of indirect jumps, accurate static construction of the control flow graph is not possible. As a result, the post-dominator information precomputed statically is imprecise and the dynamic control dependences computed are imprecise as well. In prior works [32, 27, 28] this problem is addressed by restricting the applicability of the slicing tool to binaries generated using a specific compiler which limits the applicability of the tool.

Let us illustrate the problem caused by an indirect jump using the example in Figure 7. The code snippet is shown in the first column, and the second column shows its assembly code. The switch-case statement is translated to the indirect jump: $jmp * \%eax$. Without the dynamic jump target information, in general, the static analyzer cannot figure out the possible jump targets for a indirect jump. Thus, the statically constructed CFG will be missing control flow edges from statement 4 to statements 6 and 9. This inaccurate CFG leads to an imprecise dynamic slice shown in the third column with missing control dependence $6_1 \rightarrow 4_1$.

To achieve wide applicability and precision we take the following approach in DrDebug. We implement a static analyzer based on Pin's static code discovery library – this allows DrDebug to work with any x86 or Intel64 binary. Further, we develop an algorithm to improve the accuracy of control dependence in the presence of indirect jumps. Initially we construct an approximate static CFG and as the program executes, we collect the dynamic jump targets for the indirect jumps and refine the CFG by adding the missing edges. The refined CFG is used to compute the immediate post-dominator for each basic block which is then used to dynamically detect control dependences. This leads to the accurate slice shown in the fourth column in Figure 7.

5.2 Dynamic Data Dependence Precision

Besides memory to memory dependences, we need to maintain the dependences between registers and memory to perform dynamic slicing at the binary level. Dynamic slices may include many spurious dependence edges when registers are saved/restored upon at function entry/exit. More specifically, at each function entry, registers used inside this function are saved on the stack, and later restored from the stack in reverse order when the function returns to its caller.

Consider the example in Figure 13. The first and second columns show a C code snippet and its corresponding assembly code respectively. Register eax is used in function Q, and its value is saved/restored onto/from stack at line 10/12. Considering an execution where c's value is $'t'$ at line 3, let us compute a slice for w at the first execution of statement at line 7. As variable e is used to compute w in 7_1 and its value is stored in eax, we continue to backwards traverse the trace to find the definition of register eax. As value of eax is saved/restored onto/from stack at the entry/exit of Q; we will establish data dependence edges $7_1 \rightarrow 12_1$, $12_1 \rightarrow 10_1$, and $10_1 \rightarrow 4_1$ due to eax. If only data dependences are considered, we get longer data dependence chains than needed. Since a dynamic slice is a transitive closure of both control and data dependences, we may wrongly include many spurious data and control dependences because of such data dependence chains. In the Figure 13 example, because all statements (e.g., 10_1 and 12_1) in function Q are directly or indirectly control dependent on predicate 5_1 which guards the execution of function Q, a slice for w at 7_1 will wrongly include 3_1 and 5_1 (as shown in the third column) as well as all other statements on which 3_1 and 5_1 are dependent.

C Code	Assembly Code	[Imprecise] Slice for w at line 6_1	Refined Slice

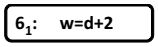
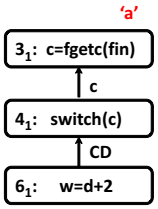
1 P(FILE∗ fin, **int** d){
2 **int** w;
3 **char** c=fgetc(fin);
4 **switch**(c){
5 **case** 'a':
/∗ *slice criterion* ∗/
6 w = d + 2;
7 **break**;
8 **case** 'b':
9 w = d − 2;
10 ... }
11}
	3 call fgetc
	mov %al,−0x9(%ebp)
	4 ...
	mov 0x8048708(,%eax,4),%eax
	jmp ∗%eax
	6 mov 0xc(%ebp),%eax
	add $0x2,%eax
	mov %eax,−0x10(%ebp)
	7 jmp 80485c8
	8 ...
			

Figure 7: Control dependences in the presence of *indirect jumps*.

Code	Assembly Code	[Imprecise] Slice for w at line 7_1	Refined Slice

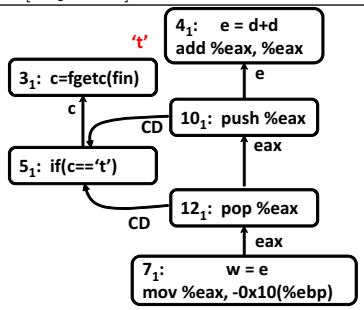
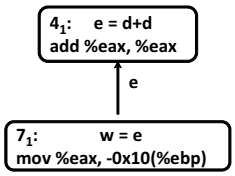
1 P(FILE∗ fin, **int** d){
2 **int** w, e;
3 **char** c=fgetc(fin);
4 e= d + d;
5 **if** (c=='t')
6 Q();
/∗ *slice criterion* ∗/
7 w=e;
8 }
9 Q()
10 {
11 ...
12 }
	3 call fgetc
	mov %al,−0x9(%ebp)
	4 mov 0xc(%ebp),%eax
	add %eax,%eax
	5 cmpb $0x74,−0x9(%ebp)
	jne 804852d
	6 call Q
	804852d:
	7 mov %eax,−0x10(%ebp)
	
	9 Q()
	10 push %eax
	...
	12 pop %eax
			

Figure 8: Spurious dependence example.

We call a pair of instructions that are only used to save/restore registers a *save/restore pair* and data/control dependences which are introduced by such pairs as *spurious dependences*. To improve the precision of the dynamic slice, we propose to precisely identify save/restore pairs and prune the spurious data/control dependence resulting from them.

Dynamically identifying save/restore pairs. One possible way is to have the compiler generate special markers for the save/restore pairs so they can be easily identified at runtime. This approach would limit the applicability of DrDebug since DrDebug is designed to work with any unmodified x86 or Intel64 binary. Therefore we use a dynamic algorithm for detecting as many save/restore pairs as possible without the help of the compiler. Our algorithm handles the following complexities caused by the compiler. First, the compilers can use either *push* (*pop*) or *mov* instruction to save (restore) the value of a register. Moreover, *push/pop* instructions are not exclusively used to save/restore registers. Second, it is not easy to know how many *push* (*pop*) or *mov* instructions are exactly used to save (restore) registers at the entry (exit) of a function. Our algorithm works as follows:

- *Statically identify potential save and restore instructions.* The first $MaxSave$ push/mov reg2mem instructions at the start of a function and the last $MaxSave$ pop/mov mem2reg instructions at the end of a function are identified as potential *save* and *restore* instructions respectively. $MaxSave$ is a tunable parameter.
- *Dynamically verify that the pairs are used to save and restore registers.* For each potential *save* instruction, we record register/memory pair and the saved value from the register. For each potential *restore* instruction, we record register/memory pairs and the restored value from the stack. An identified save/restore pair must satisfy two conditions: (1) *save* copies the value of a register r to stack location s at the entry of a function; and (2) *restore* copies the same value from s back to r at the exit of the same function. In the example of Figure 13, 10_1 and 12_1 are recognized as a save/restore pair for eax.

Pruning spurious data dependences. With recognized save/restore pairs, we prune spurious data dependence by bypassing data dependences caused by such save/restore pairs. Take the slice in the third column in Figure 13 as an example. Because $7_1 \rightarrow 12_1$, $12_1 \rightarrow 10_1$, $10_1 \rightarrow 4_1$, and 10_1 and 12_1 are recognized as a save/restore pair for eax, we bypass the data dependence chain and add a direct edge $7_1 \rightarrow 4_1$. In this way, the refined slice for w at 7_1 will not include 3_1, 5_1, and all other statements on which 3_1 and 5_1 are dependent, as shown in the fourth column.

6. IMPLEMENTATION

The implementation of DrDebug consists of Pin-based [18] and GDB-based [2] components. The Pin-based component consists of the PinPlay library and the Dynamic Slicing module. An extended Pin kit, called the PinPlay kit (available for download [5]), is used to build a pintool that does dynamic slicing, can connect to the debugger, and can also do logging/replay. The programmer interfaces with the GDB component via a command line interface [2] or a KDbg [3] based graphical interface. The GDB component communicates with the Pin-based component via PinADX [17], a debugging extension of Pin. PinPlay's logger is leveraged to generate a (region) pinball and then PinPlay's replayer can deterministically replay the execution for multi-threaded program by running off such pinball. During the replay, driven by a slice command from GDB-based [2] component, dynamic slicing module computes a slice and then PinPlay's relogger is leveraged to generate a *slice pinball*. Finally the user can single step/examine statements in slice only when PinPlay's replayer runs off the *slice pinball*.

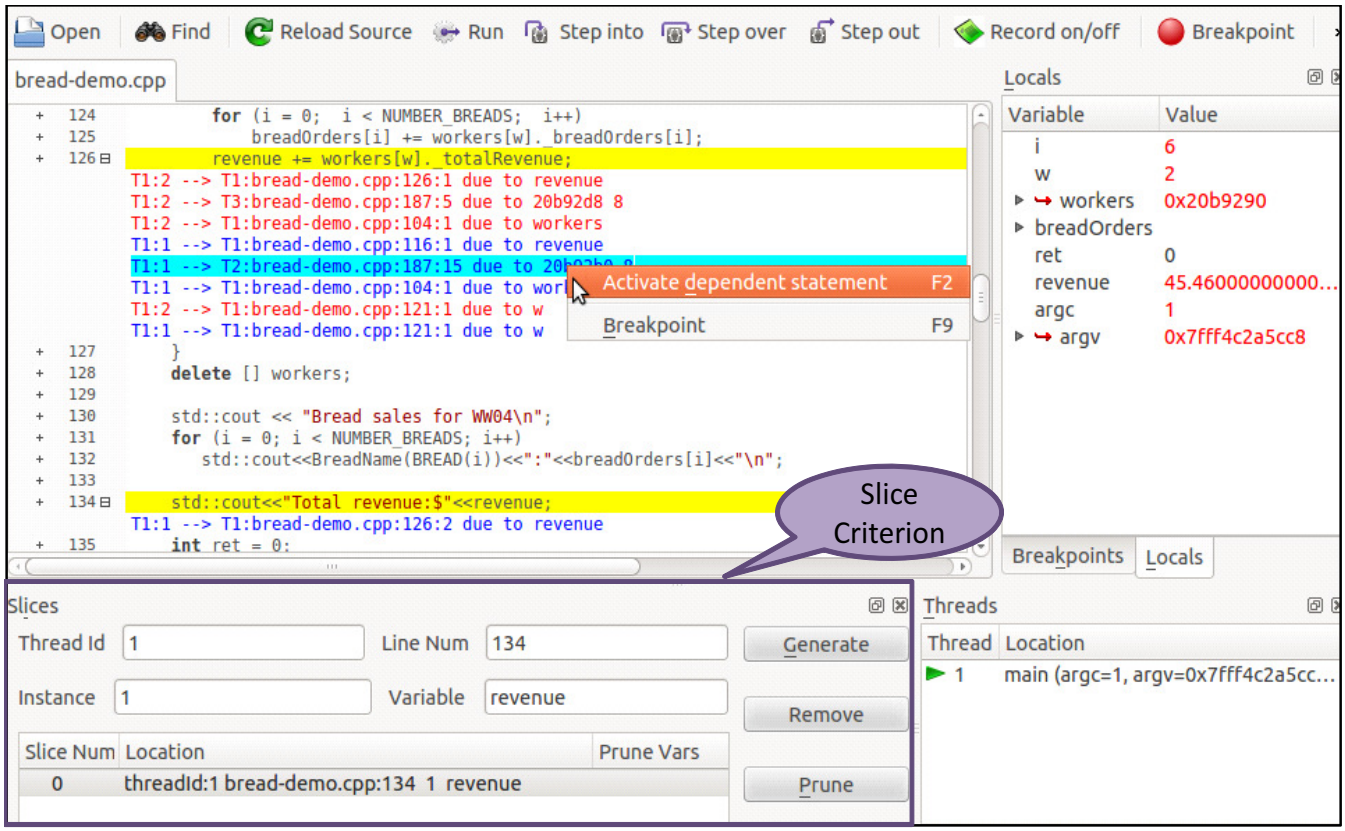


Figure 9: DrDebug GUI showing a dynamic slice.

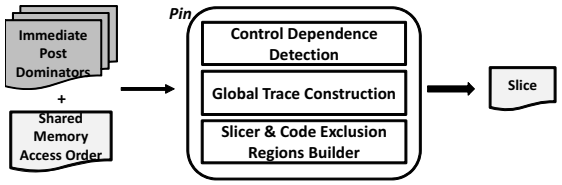


Figure 10: Dynamic Slicer Implementation.

Dynamic Slicer. The implementation of the dynamic slicing module is shown in Figure 10. We implement a static analysis module based on Pin's static code discovery library to conduct analysis to generate the control flow graph and compute the immediate post dominator information. Given immediate post dominators information, the *Control Dependence Detection* submodule [27] detects the dynamic control dependences. The *Global Trace Construction* submodule tracks individual thread traces and then constructs the global trace based on the shared memory access orders which were captured by the PinPlay's logger to enable deterministic replay. When the user issues a slice command, the *Slicer & Code Exclusion Regions Builder* submodule computes the slice by backwards traversing the global trace and then outputs the slice in two forms: a normal slice file used for slice navigation and browsing in KDbg; and another slice formatted as a sequence of code exclusion regions for use by the relogger to generate the *slice pinball*.

GUI. The extended KDbg provides an intuitive interface for selecting interesting regions for logging, and computing, inspecting and stepping through slices during replay. Figure 9 shows a screen-shot (best viewed online, in color) of our interface – all the statements in the slice are highlighted in yellow. The programmer can access the concrete inter-thread dependences and navigate backwards along dependence edges by the clicking on the Activate button of the dependent statement.

Integration with Maple. Maple [30] is a coverage-driven testing tool-set for multi-threaded programs. One of the usage models it supports helps when a programmer accidentally hits a bug for some input but is unable to reproduce the bug. Maple has two phases (i) a profiling phase where a set of inter-thread dependencies, some observed and some predicted, are recorded, and (ii) an active scheduling phase that runs the program on a single processor and controls thread execution (by changing scheduling priorities) to enforce the dependencies recorded by the profiler. The active scheduler does multiple runs until the bug is exposed.

Since Maple is based on Pin, it is an ideal candidate for integration with DrDebug. We changed the active scheduler pintool in Maple to optionally do PinPlay-based logging of the buggy execution it exposes. We had to make sure, using Pin's instrumentation ordering feature, that the thread-control done by the active scheduler does not interfere with PinPlay logger's analysis.

We have successfully recorded multiple buggy executions for the example programs in the Maple distribution. The pinballs generated could be readily replayed and debugged under GDB. We have pushed the changes we made to Maple's active scheduler back to the Maple sources [4].

Table 1: Data race bugs used in our experiments.

Program Name	Program Description	Type	Bug Source	Bug Description
pbzip2	Parallel file compressor (ver. 0.9.4)	Real	[31]	A data race on variable $fifo \rightarrow mut$ between main thread and the compressor threads.
Aget	Parallel downloader (ver. 0.57)	Real	[29]	A data race on variable *bwritten* between downloader threads and the signal handler thread.
mozilla	Web browser (ver. 1.9.1)	Real	[12]	A data race on variable $rt \rightarrow scriptFilenameTable$. One thread destroys a hash table, and another thread crashes in $js\_SweepScriptFilenames$ when accessing this hash table.

Table 2: Time and Space overhead for data race bugs with buggy execution region

Program Name	#executed instructions	#instructions in slice pinball (%instructions in slice pinball)	Logging Overhead		Replay Time(sec)	Slicing Time(sec)
			Time(sec)	Space(MB)		
pbzip2	11186	1065 (9.5%)	5.7	0.7	1.5	0.01
Aget	108695	51278(47.2%)	8.4	0.6	3.9	0.02
mozilla	999997	100 (0.01%)	9.9	1.1	3.6	1.2

Table 3: Time and Space overhead for data race bugs with whole program execution region

Program Name	#executed instructions	#instructions in slice pinball (%instructions in slice pinball)	Logging Overhead		Replay Time(sec)	Slicing Time(sec)
			Time(sec)	Space(MB)		
pbzip2	30260300	11152 (0.04%)	12.5	1.3	8.2	1.6
Aget	761592	79794 (10.5%)	10.5	1.0	10.1	52.6
mozilla	8180858	813496 (9.9%)	21.0	2.1	19.6	3200.4

7. EXPERIMENTAL EVALUATION

Case studies. We studied 3 real concurrency bugs from three widely used multithreaded programs, as detailed in Table 1. The case studies serve two purposes: (a) quantify the execution region sizes (i.e., the number of executed instructions) that need to be logged and replayed later in order to capture and fix each bug; (b) DrDebug has reasonable time and space overhead for real concurrency bugs with both whole program execution (i.e., execution region from program beginning to failure point) and buggy execution region (e.g., execution region from root cause to failure point).

Table 2 shows the time and space overhead with buggy execution region for each bug. For each bug, we captured the execution from the root cause to the failure point, and then computed a slice for the failure point during deterministic replay. The number of executed instructions is shown in the second column, while the number of instructions captured in slice pinball, as well as the percentage of number of instructions in slice pinball over total executed instruction, are presented in the third column. The time and space overhead for logging is shown in the fourth and fifth column respectively. The sixth column shows the time to replay the captured buggy region pinball, and the time for slicing is shown in the seventh column. As we can see, all concurrency bugs we studied can be reproduced with region size of 1 million instructions. Besides, the time overhead for logging, replay, and slicing is reasonable.

The time and space overhead with whole execution for each bug is shown in Table 3. For each bug, we captured the execution from the beginning of program to the failure point, simulating that novice programmers tend to capture large execution regions. As we can see, all concurrency bugs can be reproduced from the program beginning, with maximal region size of 31 million instructions. The logging, replay, and slicing time overhead is acceptable, considering the large amount of time programmers spend on debugging.

Logging and Replay. We first present results from log/replay time evaluations using 64-bit pre-built binaries with suffix 'pre' for version 2.1 of the PARSEC [10] benchmarks run on the *native* input. The goal of our evaluations was to find the logging/replay time for regions of varying sizes. We first evaluated the 4-threaded runs to find a region in the program where all four threads are created. We then chose an appropriate *skip* count for the main thread in each program which put us in the region where all threads are active. The regions chosen were not actually buggy but if they were, we can get an idea of the time to log them with PinPlay logger. For logging time evaluation, we specified regions using a *skip* and *length* for the main thread. The evaluations were done on a pool of machines with 16 Intel Xeon ("Sandy Bridge E") processors (hyper-threading OFF) and 128GB of physical memory running SUSE Linux Enterprise Server 10.

Figure 11 presents the real/wall-clock time for logging regions of varying *length* values in the main thread. We only show the results for 5 "apps" and 3 "kernels" from the PASEC benchmark suite. For each benchmark we show the logging (with bzip2 pinball compression) time in seconds for regions of length 10 million to 1 billion dynamic instructions in the main thread. The total instructions in the region from all threads were 3-4 times more than the *length* in the main thread. The times shown do not include the time to fast-forward (using *skip*) to the region but just the time reported by the PinPlay logger between the start and the end of each region. Since the logger does only minimal instrumentation before the region, the fast-forwarding can proceed at Pin-only speed. Figure 12 shows the time to replay the pinballs generated.

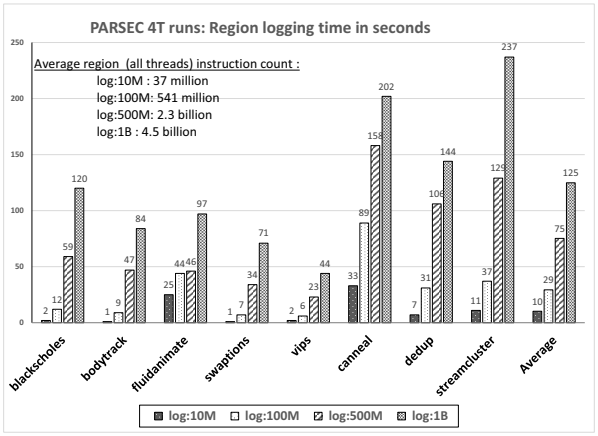


Figure 11: **Logging times** (wall clock) with regions of varying sizes for some PARSEC benchmarks ('native' input).

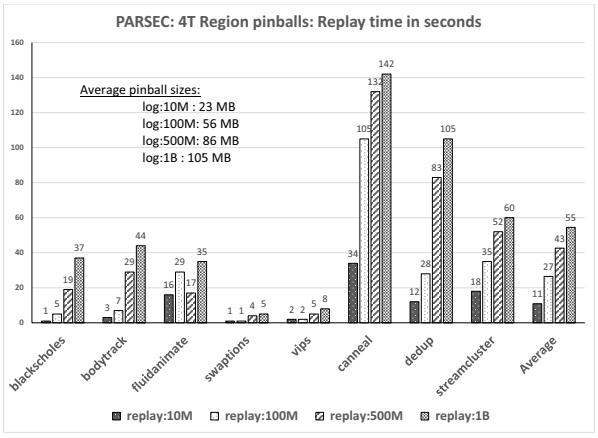


Figure 12: **Replay times** (wall clock) with pinballs for regions of varying sizes for some PARSEC benchmarks ('native' input).

Both logging and replay times take from a few seconds (length 10 million) to a couple of minutes (length 1 billion). The actual times users will see will of course depend on the length of the buggy region for their specific bug. In a study of 13 open source buggy programs reported in [21] the buggy region length (called *Window size* in the paper) was less than 10 million instructions for most programs with maximum being 18 million. We show similar analysis for some real buggy programs earlier in this section.

As described in [23], logging is more expensive than replay. However, logging will typically be done only once for capturing the buggy region. Replay will be done multiple times for cyclic debugging but since that is interlaced with user interaction and periods of user inactivity/thinking breaks, the replay overhead will be hardly noticeable (at least in our experience).

Finally, note that the pinballs (in tens to hundreds of MB in size) are small enough to be portable, so a buggy pinball can be transferred from one developer to another or from a customer site to a vendor site. The pinball size is *not* directly

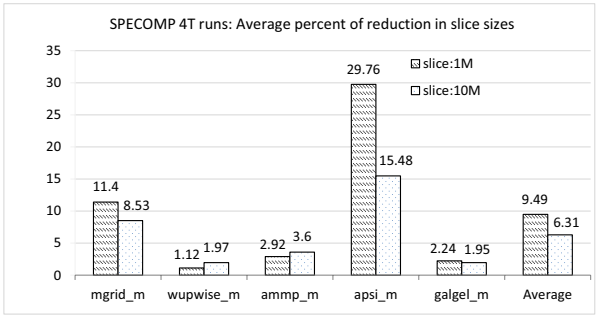


Figure 13: **Removal of spurious dependences** - Average percentages of reduction in slice sizes over 10 slices for regions of length 1 million and 10 million dynamic instructions: SPECOMP (medium, test input).

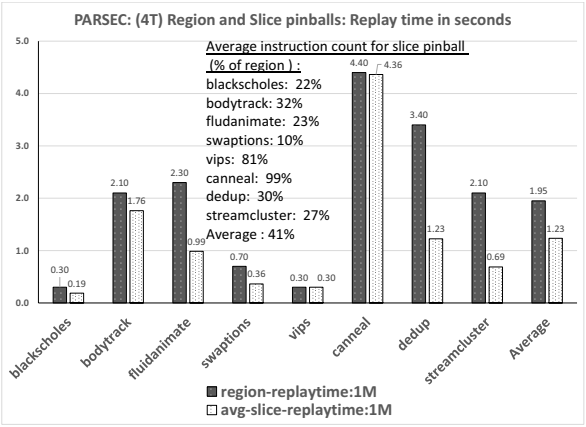


Figure 14: **Execution slicing** - Average replay times (wall clock) for 10 slices for regions of length 1 million dynamic instructions: PARSEC ('native' input).

a function of region length but depends on memory access pattern and amount of thread interaction [23]. Hence pinballs for regions with length more than 1 billion instructions need not be substantially larger. In fact, sizes of pinballs for the entire execution of the five programs are in the range 4MB–145MB, much smaller than most region pinball sizes.

Slicing overhead and precision. There are two components to the slicing overhead: the time to collect dynamic information needed for efficient and precise dynamic slicing and the time to actually perform slicing. For the 8 PARSEC programs we tested, the average dynamic information tracing time for region pinballs with 1 million instructions (main thread) was 51 seconds. Once collected, the dynamic information can be used for multiple slicing sessions as PinPlay guarantees repeatability. For evaluating slicing time we computed slices for the last 10 read instructions (spread across five threads) for each region pinball. For the regions with 1 million instructions (main thread), the average size of the slice found was 218 thousand instructions and the average slicing time was 585 seconds.

We also measured the reduction in dynamic slice sizes achieved by pruning spurious dependences by identifying save/restore pairs. As the sizes of 10 dynamic slices for the 8 PARSEC programs are only slightly influenced with the spurious dependences prune, we omit the results here.

Instead, we evaluated the effect of spurious dependences prune with five programs (ammp, apsi, galgel, mgrid, and wupwise) from SPECOMP 2001 benchmarks [9]. Figure 13 shows that, on average, dynamic slice sizes are reduced by 9.49% (6.31%) for 1 million (10 million) instructions region pinballs ($MaxSave$ is set to 10).

Execution slicing. When an execution slice is replayed using the slice pinball, the execution of code regions not included in the slice is skipped making the replay faster. In Figure 14 we present the average replay time for 10 execution slice pinballs and the replay time for original, unsliced, pinball for regions of length 1 million instructions (main thread). Also included are average count of dynamic instructions in the slice pinballs as a percentage of total instructions in the full region pinball. As we can see, on average only 41% of dynamic instructions from a region pinball are included in an average slice. This makes the replay 36% faster on average. It also shows that the programmer will need to step through the execution of only 41% of executed instructions to localize the bug. Thus debugging via slice pinball and execution slices enhances debugging.

8. RELATED WORK

Debugging. Newer versions of GDB have two related features: native record/replay and reverse debugging [1]. The record/replay feature appears to work by recording every instruction along with the changes it makes to registers and memory. This could require a huge amount of storage and cause a large slowdown. UndoDB-gdb [7] does a much more efficient implementation of the record/replay and reverse debugging features. UndoDB "uses a 'snapshot-and-replay' technique, which stores periodic copy-on-write snapshots of the application and non-deterministic inputs"(quoted from [7]). TotalView debugger [6] has support for reverse debugging with ReplayEngine. It apparently works by forking multiple processes at different points in the recorded region and attaching to the right process on a 'step back' command.

The checkpoints in all these debuggers are specific to a debug session and do not help with cyclic debugging. The real purpose of the reverse debugging commands is to find the points in the execution that affect a buggy outcome. Dynamic slicing is a more systematic way to find the same information that allows more focussed backward navigation. We believe reverse debugging can be supported in the DrDebug tool-chain by recording multiple pinballs and then replaying forward using the right pinball. Doing this using PinPlay's user-level check-pointing feature can be much more efficient than using operating system features.

VMWare supported replay debugging in their "Workstation" product between 2008 and 2011 [8]. Recording could be done either using a separate VMWare Workstation user-interface or with Microsoft's Visual Studio debugger. The debugging could be done in Visual Studio. The recording overhead was extremely low because only truly non-reproducible events were captured by observing the operating system from a virtual machine monitor. The program had to be run on a single processor though. That could make capturing certain multi-threaded bugs very hard. Also, the replay-based debugging worked only with the virtual machine configuration where it was created. The PinPlay framework we use does not require any special environment such as the virtual machine. Recording can be done in a program's native environment and the recording can be replayed/debugged on any other machine.

Whyline [13] allows programmers to ask "why?" and "why not?" questions about program outputs and provides possible explanations based on program analysis, including static and dynamic slicing. Compared with DrDebug, Whyline has several drawbacks. First, Whyline only supports post-mortem analysis, while DrDebug supports both backwards reasoning along dependence edges as well as forwards single-stepping the slice in a live debugging session. Second, since it does not integrating a record/replay system, Whyline does not support deterministic cyclic debugging. Third, programmers can only pick questions regarding program outputs only, while with DrDebug, programmers can compute slice for any interested variables/registers.

Execution Reduction. Several existing works on execution reduction [34, 25, 16, 11] either reduce the tracing overhead during replay [34, 25] or replay overhead [16, 11]. In [34] and [25] authors support tracing and slicing of long-running multi-threaded programs. They leverage meta slicing to keep events that are in the transitive closure of data and control dependences with respect to the event specified as slicing criterion. However, the slicing criterion can only be events (e.g., I/O) captured during the logging phase. On the other hand, DrDebug can reduce the replay pinball for any variable. Lee et al. [16] proposed a technique to record extra information during logging and then leveraged it to reduce the replay log in unit granularity based on programmers' annotation of unit. DrDebug's execution region enables programmers to only log and fast forward to the reasonable small buggy region during replay. Thus, DrDebug can reduce the region pinball to a slice pinball at finer granularity without requiring unit annotations by the programmer. LEAN [11] presents an approach to remove redundant threads with delta debugging and redundant instructions with dynamic slicing while maintaining the reproducibility of concurrency bugs. However, the overhead of delta debugging can be very high as it requires repeated execution of replay runs. More importantly, none of these works support stepping through a slice in a live debugging session.

Program Slicing. Static slicing has been extended for concurrency programs [15, 19]. Tallam et al. extended dynamic slicing to detect data races [24]. Weeratunge et al. [26] presented dual slicing by leveraging both passing and failing runs. However, neither approach is designed to be integrated with a record/replay system, where we can reuse the shared memory access orders. Meanwhile, our dynamic slicing algorithm is highly precise via CFG refinement using dynamic jump targets and bypassing of spurious data dependence caused by save/restore pairs.

9. CONCLUSIONS

Cyclic debugging of multi-threaded programs is challenging mainly due to run-to-run variation in program state. We have developed a set of program record/replay based tools to address the challenge. We also provide tools for creating dynamic slices, check-pointing just the statements in a given dynamic slice, and navigating through the slice recording. The tools work in conjunction with a real debugger (GDB) with a graphical user interface front-end (KDbg). By focusing on a buggy region instead of the entire execution, the time for recording/replaying/dynamic slicing can be quite reasonable making the tools practical.

10. ACKNOWLEDGMENTS

This research is supported by NSF grant CCF-0963996 and a grant from Intel to the Univ. of California, Riverside.

We thank Paul Petersen, Gilles Pokam, Chuck Yount, Jim Cownie, and the anonymous reviewers for feedback on the early drafts of this paper. Thanks to Ady Tal and Omer Mor for help with PinPlay; Tevi Devor, Nafta Shalev, and Sion Berkowits for help with Pin; Jie Yu and Satish Narayanasamy for Maple consultation, and Moshe Bach, Robert Cohn, Geoff Lowney, and Peng Tu for supporting this work over the years.

11. REFERENCES

[1] Gdb and reverse debugging. http://www.gnu.org/software/gdb/news/reversible.html.

[2] Gdb page. http://www.gnu.org/software/gdb/.

[3] Kdbg page. http://www.kdbg.org/.

[4] Maple sources. https://github.com/jieyu/maple.

[5] Program record/replay (PinPlay) toolkit. http://www.pinplay.org.

[6] Reverse debugging with replayengine. http://www.roguewave.com/products/totalview/replayengine.aspx.

[7] Undodb page. http://undo-software.com/product/undodb-man-page.

[8] Vmware's replay debugging offering. http://www.replaydebugging.com/.

[9] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *In Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[10] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[11] J. Huang and C. Zhang. Lean: simplifying concurrency bug reproduction via replay-supported execution reduction. OOPSLA '12, pages 451–466, 2012.

[12] D. Jeffrey, Y. Wang, C. Tian, and R. Gupta. Isolating bugs in multithreaded programs using execution suppression. *Software: Practice and Experience*, 41(11):1259–1288, 2011.

[13] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. *ICSE'08*, pages 301–310.

[14] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[15] J. Krinke. Context-sensitive slicing of concurrent programs. ESEC/FSE-11, pages 178–187, 2003.

[16] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward generating reducible replay logs. PLDI'11, pages 246–257.

[17] G. Lueck, H. Patil, and C. Pereira. PinADX: an interface for customizable debugging with dynamic instrumentation. In *CGO'12*, pages 114–123, 2012.

[18] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05*, pages 190–200, 2005.

[19] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, Nov. 2006.

[20] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS'06*, pages 216–227, 2006.

[21] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA'05*, pages 284–295, 2005.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*, pages 89–100.

[23] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. CGO'10, pages 2–11.

[24] S. Tallam, C. Tian, and R. Gupta. Dynamic slicing of multithreaded programs for race detection. In *ICSM'08*, pages 97–106, 2008.

[25] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. ISSTA '07, pages 207–218, 2007.

[26] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. ISSTA'10, pages 253–264, 2010.

[27] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA'07*, pages 185–195.

[28] B. Xin and X. Zhang. Memory slicing. ISSTA'09, pages 165–176, 2009.

[29] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. ISCA'09, pages 325–336, 2009.

[30] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA'12*, pages 485–502, 2012.

[31] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. *ACM SIGPLAN Notices*, 45(3):179–192, March 2010.

[32] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *PLDI'06*, pages 169–180, June 2006.

[33] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *ICSE'03*, pages 319–329, May 2003.

[34] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. SIGSOFT '06/FSE-14, pages 81–91, 2006.