DPAC: An Infrastructure for Dynamic Program Analysis of Concurrency Java Programs

Yanyan Jiang, Chang Xu*, Xiaoxing Ma*
Nanjing University, State Key Laboratory for Novel Software Technology, 210093 Nanjing, China
jiangyy@outlook.com, {changxu, xxm}@nju.edu.cn

ABSTRACT

Concurrency programs are hard to test or debug due to their non-deterministic nature. Existing dynamic program analysis approaches tried to address this by carefully examine a recorded execution trace. However, developing such analysis tools is complicated, requiring to take care of many tedious implementation details, and comparing and evaluating different analysis approaches are also subject to various biases, due to lack of a common base platform. This motivates us to design DPAC, an infrastructure that support in building dynamic program analysis tools for concurrency Java programs. DPAC takes events and their various processing mechanisms as its underlying model to facilitate monitoring and manipulation of program executions as required by dynamic program analysis. Various analysis tools can be implemented by customizing their required event types and processing mechanisms. We show two concrete case studies how our DPAC helps building existing dynamic program analysis approaches, as well as tuning subtle implementation details for supporting customized function implementation and code transformation.

Categories and Subject Descriptors

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

General Terms

Algorithm, Performance, Reliability

Keywords

Concurrency, dynamic program analysis, platform support

1. INTRODUCTION

Concurrency programs often misbehave because of unintended thread scheduling. Testing, debugging or even reproducing these concurrency bugs are challenging because these bugs might be triggered only by very specific thread interleaving. Currently, such testing and debugging issues are largely addressed by static and dynamic *program analysis* techniques.

On the one hand, traditional *static program analysis* techniques working on the abstract model of programs (e.g., data flow analysis and model checking), could be extended for analyzing concurrency programs [7, 8]. However, the input space plus the interleaving space of a concurrency program is double-exponential. Finding a proper level of abstraction is still considered very challenging.

On the other hand, facing such difficulties, *dynamic program analysis* offers an alternative by restricting the scope to only examining the interleaving space. The most fundamental dynamic analysis tool is *deterministic replay* [6,9,10,13], which provides the time-travel debugging ability. On the basis of deterministic replay, *heuristic random testing* tools are built. They have been proven useful in finding access anomalies such as data races and atomicity violations [11, 17], or in efficiently exploring the interleaving space [5, 15]. To achieve even higher coverage, *predictive trace analysis* (PTA) techniques systematically explores the interleaving space of a recorded execution under certain scheduling constraints, such as a causal model [18, 20] or context switching bound [14]. More and more such novel technologies and improvements for dynamically analyzing concurrency programs are emerging, and we are motivated to design and to implement a middleware supporting for the following reasons:

1. Implementation of such program analysis tools involves large engineering efforts and tedious low-level details. It would be too costly if we build up the prototype from scratch each time when we want to evaluate an approach or to launch some experiments. A middleware would significantly *boost this prototype development cycle*.

2. Evaluation of the existing work without bias is sometimes difficult because efficiency of algorithms might be extremely sensitive to the underlying implementation and optimizations (e.g., a static alias analysis would accelerate the deterministic replay by one or two magnitude [13]). A middleware would help up us *alleviating such kind of biases* by implementing different approaches under the common platform.

A dynamic program analysis tool typically poses two kinds of demands for such a middleware. Firstly, to accelerate the software development cycle, adequate API abstraction should be provided. At the same time, capability for users to perform arbitrary customized program instrumentation should also be given. Note that these two kinds of demands might be both required even in a single tool. Hence the main challenging of designing such a middleware is how to integrate these two suites of APIs in a single system.

In this paper, we show that these two seemingly conflicting goals could be simultaneously achieved by designing and implement-

*Corresponding authors.

Middleware 2013 Doctoral Symposium, December 10, 2013, Beijing, China.

ing DPAC, an infrastructure for conveniently and flexibly developing dynamic program analysis tools of concurrency Java programs. DPAC is designed according to the high-level perspective that building dynamic program analysis systems for concurrency programs mainly addresses the problem of *execution monitoring* and *execution manipulation*, since dynamic analysis mainly targets on analyzing execution traces. We found the *event processing model* is adequate for fulfilling such demands: Concurrency programs are instrumented to dispatch events at runtime, and the analysis tool is implemented as a series of event handlers. DPAC provides both libraries supporting frequently used events and their handlers, as well as program instrumentation interfaces for developing customized event mechanisms.

DPAC is consisted of four major components: the engine, the analyzer, the transformer and the runtime, as shown in Figure 1. In DPAC, dynamic analysis tool is implemented as collaborating event handlers in the runtime, processing the events generated by the engine and the bytecode modified by the transformer. Analytic results provided by the analyzer could also be exploited by the other parts of the system. To further aid the tool development, DPAC provides APIs on various levels of abstraction for static program analysis, program instrumentation and event handling.

We believe our main contribution is design and implementation of DPAC that benefits quality and productivity of developing dynamic program analysis tools. Several preliminary evaluation results support our claim, and we have already started developing research dynamic program analysis prototypes based on our DPAC infrastructure.

The rest of the paper is organized as follows. Section 2 introduces the system design of DPAC that how do we decouple the problem of dynamic analysis of concurrency programs. Section 3 supplements some implementation decisions for achieving high efficiency in DPAC. Section 4 evaluates DPAC by exhibiting how two research prototypes are built upon DPAC. Section 5 discusses the innovation of DPAC over the prior work, while Section 6 draws our conclusion and future directions.

2. SYSTEM DESIGN

2.1 Design Overview

DPAC aims at supporting dynamic program analysis tools for analyzing concurrency Java programs. Such tools typically pose the need of the following types of supports:

1. The ability to monitor the execution of a concurrency program with minimal perturbation. Typically, only a small subset of all actions performed by the program is interested, and this set is often computed by static program analysis.

2. The ability to manipulate the behavior of a concurrency program, such as controlling the threads to perform context switch at a specific program point, or modifying the return value of a memory access.

To unify the procedure of execution monitoring and execution manipulation, we design the high-level model of DPAC to be an event processing system. Running threads could generate synchronous[1] events simultaneously in parallel, while the task of program state monitoring and execution manipulation are both implemented in the event handler: Program states are encapsulated in the parameters of an event, and are altered on the instructions of the event handlers. By forcing components in DPAC event-driven, DPAC successfully decouples the design and implementation of modules in the system, at the same time providing flexibility for future extensions.

[1]A thread is blocked after generating an event, and is resumed when the event handler returns.

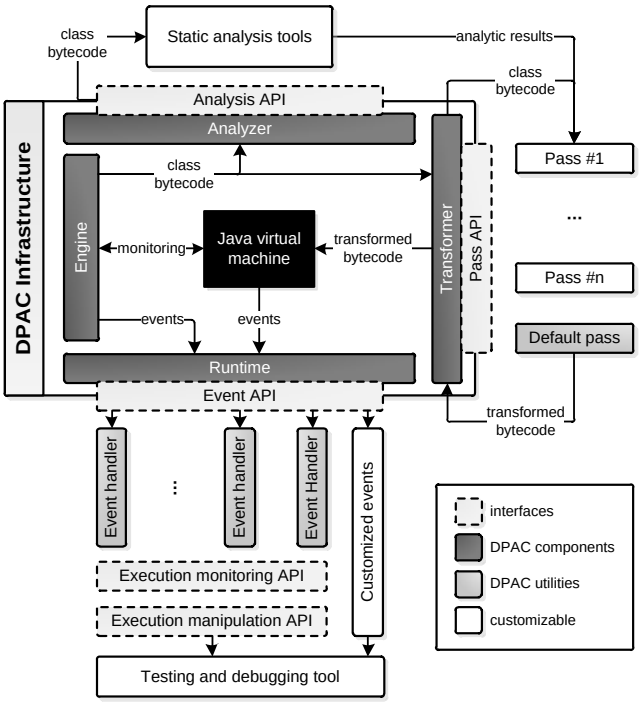

Figure 1: DPAC architecture

We also design DPAC to provide flexible libraries for both low-level program instrumentation and high-level function implementation. The low-level program instrumentation API wraps bytecode manipulation library to offer customizable event generation and handling, while the high-level libraries provide frequently used routines in dynamic program analysis such as shared memory access monitoring and schedule controlling. In most cases, the high-level libraries would be sufficient for implementing dynamic analysis tools, hence accelerating the development cycle. For those tools requiring code manipulation or performance tuning, low-level interface could be used.

2.2 DPAC Architecture

DPAC divides the task of event generation and handling into four major components: the engine, the analyzer, the transformer and the runtime, as shown in Figure 1. The engine serves as the control center of the DPAC system. It captures low-level virtual machine events such as class loading, thread creation and virtual machine shutdown, and is responsible for routing these events to their corresponding handlers. The analyzer performs static analysis on the source code, producing intermediate results (e.g., program slicing information) used by the transformer and the runtime. The transformer provides API for program instrumentation. The instrumentation procedure is divided into several passes. At each pass, the transformer traverses all bytecode instructions in every method of a class for customized manipulations. Finally, the runtime is an interface layer between the instrumented bytecode and the actual program analysis tool implementation. It decodes events and then invokes the corresponding event handlers. Detailed designs of these components are described below.

2.2.1 The Engine

The engine starts along with the code being analyzed. It could be regarded as the low-level control center capturing virtual machine events and routing them to the other components in the system. It captures the following event types:

1. Virtual machine initialization and termination. These events are critical because classes in DPAC must be initialized before the main class is loaded, and finalization work must be performed before the virtual machine is shut down.

2. Thread creation and termination. Thread is the primitive building block of a concurrency program. Since many dynamic analysis tools trace the status of threads, the engine should be aware of these events.

3. Class loading. To track more detailed program execution, one would need to perform customized instrumentation to the program. Class loading event is generated each time a class is loaded in the JVM for such instrumentations.

4. Other useful events such as method invocation, thread suspension and garbage collection.

In DPAC, except for the class loading events which are routed to the transformer, all other events are routed to the runtime component.

2.2.2 The Analyzer

Static analysis results are often beneficial for dynamic analysis tools. The analyzer is designed to provide an interface between the static analysis programs and the dynamic analysis tools implemented in DPAC.

Currently, the analyzer is under development and only has limited function. Each static analysis algorithm generates sets of bytecode instructions (uniquely identified by the class, the method and the line number) indicating that instructions in the same set share some common property (e.g., memory access instructions are always correctly synchronized). These analysis results are stored in the text format, and are passed to the transformer and the runtime by the engine. This is sufficient for our current research projects, but we have planned to extend its flexibility in the future.

2.2.3 The Transformer

Sometimes, events provided by the engine are not sufficient or unsuitable for implementing specific program analysis tools. User might want to perform customized instrumentations on the class bytecode. However, doing such low-level instrumentation using bytecode transformation libraries is complicated and error-prone. The transformer greatly simplifies this procedure by dividing the whole instrumentation process into several passes, as illustrated in Figure 2. Every instruction of every method is traversed in each pass, and user is able to perform instrumentations on these instructions. The pass API helps user (1) inserting event dispatchers according to the instruction context; (2) doing arbitrary insertion, deletion or modification of instructions; (3) managing method meta-data such as stack frame size. Program analysis tools often use the transformer to insert event generation code to invoke stubs in the runtime.

The transformer includes many commonly used transformation passes. For example, user could override methods in `MemoryAccessTransformer` to have customized instrumentation for shared memory accesses. Another example is `ClassSignatureTransformer` which replaces all occurrences of a class signature with a specified one. This is extremely useful for in instrumenting system libraries.

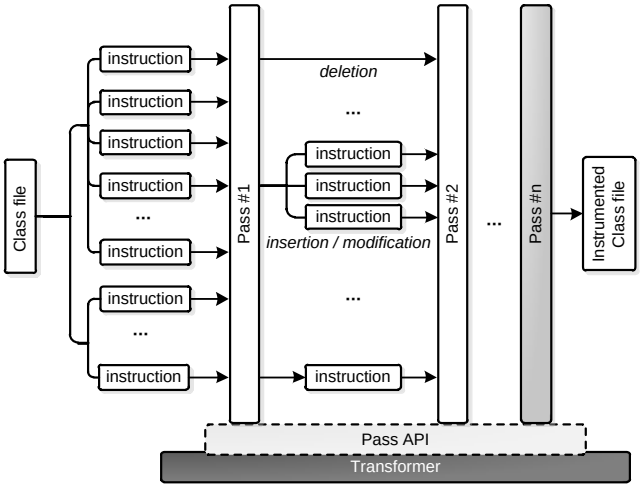

Figure 2: DPAC passes

After all the customized passes finish their execution, DPAC instruments the program by several standard passes. The code is transformed to generate monitor events and memory access events because these events are interested by most dynamic analysis tools.

2.2.4 The Runtime

The runtime connects the DPAC internals with the actual program analysis tool implementation. It is the sink of all events except for the class-loading ones. Each program analysis tool is implemented as a subclass of `RuntimeAdapter` implementing the concrete event handlers of the following types:

1. Handlers for events provided by DPAC. For example, the method overriding `threadCreation(Thread t)` will be invoked each time when a thread is created, and the corresponding thread object will be passed as the argument.

2. Handlers for customized events. To reduce the performance penalty of event handling to a minimum, these handlers are implemented as plain Java methods, and are directly invoked by the instrumented bytecode.

Like the transformer, the runtime also integrates rich predefined utilities. An example is a schedule controller (SC). SC captures all events that might alter a thread's execution path, and the thread generating that event is immediately blocked by SC. At the time when all threads are suspended, SC asks user to decide which thread is about to be executed next. Many dynamic analysis tools (e.g., replayer in deterministic replay and interleaving space exploration engine) could be easily built upon SC.

2.3 Example: Implementing Random Testing

To illustrate how the components interact with each other in DPAC, consider an example that we are implementing the race directed random testing described in [17]. The program source is firstly sent to the analyzer for a static lock-set analysis [16]. The static analysis finds pairs of shared memory accesses which are potentially data races. These potential races are tried to be confirmed as real races by a random scheduler. The random scheduler works as follows. The results of static analysis are firstly delivered to the

```
void scheduler() {
  while (threadCount > 0) {
    allSleep.acquire();
    // resume a blocked thread
    synchronized {
      tid = pickUp(nextEvent);
      blockCount --;
      getToken(tid).release();
    }
  }
}
void request(Event e) {
  long tid = Thread.currentThread.getId();
  synchronized {
    nextEvent.put(tid, e);
    blockCount++;
    if (blockCount == threadCount) {
      allSleep.release();
    }
  }
  getToken(tid).acquire(); // block current thread
  nextEvent.put(tid, null);
}
...
// pause thread before interested events
void beforeRead(int cl, Object ref, int field) {
  request(new MemoryEvent(READ, cl, ref, field));
}
void beforeMonitorEnter(Object ref) {
  request(new MonitorEvent(ENTER, ref));
}
...
// keep track of amount of threads in the system
synchronized void beforeFork(Thread t) {
  threadCount++;
}
synchronized void threadEnd(Thead t) {
  threadCount--;
}
```

Figure 3: Code snippet from a schedule controller

transformer at runtime. At class-loading time, the transformer instruments every potentially racing shared memory access. A small piece of event dispatching code is inserted before all such memory accesses. The runtime could now pause the thread before a potentially racing memory access is about to be executed simply by suspending currently running thread in those event handlers. With the assistance of schedule controller which is already provided by the DPAC infrastructure, the race directed random testing is implemented.

3. IMPLEMENTATION

We implemented the DPAC infrastructure on the Java platform. Our implementation does not involve modification of the Java virtual machine, and could run on any Java virtual machine implementing the JVM tool interface (JVM TI) [2].

The engine is implemented as an agent of JVM TI in C++. All other components are pure-Java implemented. The engine captures several JVM TI events such as VM initialization, thread creation, thread termination, class loading and VM termination. The engine routes these events to other components of DPAC. Class-loading events are routed to the transformer, while the others are routed to the runtime for executing the user-specified handlers. Since event generation might be very frequent, the event handlers reside in the same Java virtual machine as the engine and the code being analyzed. Event dispatches are implemented by method invocation, which could be further optimized by the just-in-time compiler. This also allows multiple threads handling events simultaneously in parallel. To ensure the event handler classes are not erroneously instrumented, DPAC provides an annotation `@NotInstrument` indicating that a class is belonging to the runtime component and should not be instrumented.

The analyzer is currently only several utility functions for generating uniformed analysis results. Users usually invoke Soot [19] for static analysis, and the outputs of Soot are mapped to our analysis result format by the analyzer API.

The transformer is an RPC server which could be deployed on a remote machine. Such design isolates the execution of transformer in a separate JVM for name space concerns. The transformer is initialized by the engine with the results of the analyzer. Each time JVM receives a class loading request, the class bytecode is sent to the transformer for instrumentation. This approach enables instrumentation of on-the-fly created classes and system libraries. Our transformation pass model is implemented on the top of the ASM bytecode framework [1]. After completion of the last pass, the transformer sends back the transformed bytecode to the engine. Because bytecode transformation is a time-consuming task, the transformed bytecode is then cached by the engine: Next time when the engine encounters a class with the same signature and bytecode fingerprint, cached bytecode will be immediately used rather than requesting the transformer to perform a redundant transformation.

The runtime captures the events generated either by the engine or by the transformed bytecode through method invocation. Some kinds of event frequently used by the dynamic analysis tools are already provided by the runtime. For example, the runtime receives `beforeRead`, `beforeWrite`, `afterRead` and `afterWrite` events each time when a thread is accessing the shared memory. Every such event contains arguments indicating the class identifier, the field identifier and the object reference of the shared memory access. Dynamic analysis tools relying on these kinds of events could simply override the default handlers for these events. Otherwise, if one does not specify such handlers, just-in-time compiler of JVM will automatically inline these empty default event handlers at zero performance penalty.

4. EVALUATION

We claimed that DPAC provides both enough abstraction accelerating the development cycle of dynamic program analysis tools, as well as great flexibility to implement tools with customizable instrumentation requirements. Preliminary evaluation results are delivered in this section. To validate the first claim, we demonstrate how to implement a thread schedule controller on our DPAC infrastructure in Section 4.1. To validate the second claim, we present our experimental results of ongoing work of an optimized deterministic replay algorithm in Section 4.2.

4.1 Implementing Schedule Controller

We show how to implement a thread schedule controller discussed in Section 2.2.4 under the framework of DPAC within a few lines of code. This would be useful in implementing random testing tools such as RaceFuzzer [17] and PCT [5]. The code snippet is shown in Figure 3.

As illustrated, the implementation is quite straightforward that a thread pauses itself each time when it is about to execute a shared-state related event. When all threads are slept, the scheduler decides which one to run by invoking `pickUp`. It is clear that DPAC frees the user from complicated details of low-level instrumentation, and only needs to specify what to do at the interested events. It is easy to

```
@Override public InstrumentResult instrument(
    Instruction ins) {
  Instruction[] prev = null, next = null;
  requestStackUse(6);   // claim stack growth
  refVar = localVar(1); // allocate local variable
  Operand operand = new Operand(ins);
  ...
  next = new Instruction[] {
    // ------- prepare event arguments -------
    ALOAD(refVar),          // object reference
    LOADOFFSET(),           // field identifier
    LOAD(operand, valVar), // return value
    // --------- dispatch the event ----------
    EVENT("cacheFetch",     // event name
      "O", "I", operand,    // event argument list
      cacheClass            // event receiver
    ),
  };
  ...
  return new InstrumentResult(prev, next);
}
```

Figure 4: Code snippet from our deterministic replay implementation

extend this approach to implement other dynamic program analysis tools such as deterministic replay and predictive trace analysis. We have already implemented several such tools in the framework of DPAC.

4.2 Implementing Efficiency-critical Deterministic Replay

We now show that DPAC also provides flexibility of low-level instrumentation. We use our ongoing research work on deterministic replay as the example. Recognizing that read actions dominate the shared memory accesses for most real-world programs, we decide to focus on reducing the logging cost of these read actions. However, low record-time cost and high replay trace fidelity are hardly simultaneously achieved. We are trying to find a good trade off which substantially improves the record-time efficiency with only a mild loss of replay guarantee.

In this research work performance is critical because its efficiency directly impacts its usefulness. Thus we use the analyzer to exploit the results of static alias analysis, and use low-level instrumentation API provided by the transformer for bytecode manipulation. Figure 4 shows a code snippet from the transformer pass instrumenting the heap read in our current implementation.

Some preliminary experiment results are shown in the Table 1. These results show that our work is very efficient at log size compared with the state-of-are work Stride [21]. However, there are still magnitudes of time overhead gaps between our experimental results and the one reported in the Stride paper, even though our basic idea bears lots of similarity with Stride. We believe this is caused by the low-level implementation and several optimizations mentioned in the paper, but none of them is reproducible. This refers to our second motivation: To fairly compare between approaches, we need to implement them under a common base platform. DPAC serves such a purpose. We will implement related work for fair comparison on DPAC in the near future.

5. RELATED WORK

Implementation of a dynamic program analysis tool would be complicated without appropriate middleware support. For example, deterministic replay tools such as LEAP [9] and Stride [21] are built directly upon the Soot Java code optimization framework [19]. Soot provides handy support for static program analysis (e.g., escape analysis and race-free analysis) and optimization, but is relatively inconvenient at source code instrumentation. It is also difficult for Soot to instrument system libraries and dynamically generated classes, which could easily be done at bytecode or instruction level.

Nirvana platform [3] gives simple and compact API to observe instruction-level program behavior at runtime. Deterministic replay tool iDNA is implemented on Nirvana for demonstration. However, targeting at low-level instruction prohibits performing static analyses because much of program information is lost during the binary translation. Tools based on Nirvana reports significantly higher overhead than those work with heavy static analysis optimizations [13].

Java Path Finder (JPF) [4] offers flexible program instrumentation and verification capability via simulated execution of Java bytecode, which can be considered as a separate implementation of JVM targeting at execution profiling and control. Since optimization is not its primary concern, JPF operates in interpretation mode hence does not have comparable performance as production JVM implementations, and does not provide efficient way to implement light-weighted instrumentation probes. Although JPF is highly mature and customizable, its overhead would be too much for performance-critical scenarios such as production application runs.

Compared with the work above, DPAC gives full integration of low-level and high-level APIs for implementing dynamic program analysis, as well as interfaces connecting static program analysis tools. High-level interfaces are very convenient for fast prototyping, which could be used to validate new research ideas. Efficiency-critical applications could also be implemented by the bytecode transformation API, which works well with Oracle's production JVM implementation. With the aid of DPAC infrastructure, programmer could focus on "what to do" in developing program analysis tools, rather than being drawn into the error-prone low-level details.

6. CONCLUSION AND FUTURE WORK

In conclusion, DPAC is a light-weight event processing framework for developing dynamic program analysis tools of concurrency Java programs. We have shown that various such tools could be easily built on DPAC. Furthermore, DPAC also provides developers the full capability of performing arbitrary low-level bytecode transformation.

Still, this work is not complete. Our current research work mainly focuses on the deterministic replay. Only a few random testing and predictive trace analysis work are implemented on the DPAC framework. Furthermore, it would be difficult for our current DPAC implementation to support advanced software model checking tools such as dynamic symbolic execution [12]. In the future, we might also find difficulties implementing rapidly emerging program analysis techniques due to limited capability provided by DPAC. We hope to incrementally improve the design and implementation of DPAC during our subsequent research of testing and debugging for the concurrency programs.

Acknowledgements

This research was partially funded by National High-tech R&D 863 Program (No. 2012AA011205) and National Natural Science Foundation (Nos. 61100038, 91318301, 61361120097) of China.

Benchmark	*Our work*		*Data Reported in* [21]	
	Overhead (X)	*Log Size(MB/s)*	*Overhead (X)*	*Log Size(MB/s)*
Avrora	5.7	0.45	10.58	257
Batik	8.3	1.53	0.08	0.01
Tomcat	9.1	0.51	0.09	0.13
Tsp	4.3	1.30	1.54	39.8
Moldyn	35	6.74	1.50	27.3

Table 1: Preliminary evaluation results of our novel deterministic replay algorithm

Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (No. NCET-10-0486).

References

[1] ASM toolkit for bytecode manipulation. `http://asm.ow2.org/`.

[2] JVM tool interface. `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[3] 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 *Proceedings of the 2nd international conference on Virtual execution environments*, VEE, 2006.

[4] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - second generation of a java model checker. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[5] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS, 2010.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI, 2002.

[7] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP, 2003.

[8] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, 2005.

[9] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE, 2010.

[10] J. Huang, C. Zhang, and J. Dolby. CLAP: recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2013.

[11] P. Joshi, M. Naik, C. S. Park, and K. Sen. CalFuzzer: an extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV, 2009.

[12] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2012.

[13] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, 2012.

[14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI, 2008.

[15] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, 2012.

[16] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP, 2003.

[17] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2008.

[18] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, 2012.

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: a java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON, 2010.

[20] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM, 2009.

[21] J. Zhou, X. Xiao, and C. Zhang. Stride: search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE, 2012.