Low-Overhead Bug Fingerprinting for Fast Debugging

Cristian Zamfir and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. There is a gap between the information available at the time of a software failure and the information actually shipped to developers in the corresponding bug report. As a result, identifying the cause of the bug based on this bug report is often difficult. To close this gap, we propose *bug fingerprints*—an augmentation of classic automated bug reports with runtime information about how the reported bug occurred in production.

Classic automated bug reporting systems contain at most a coredump that describes the final manifestation of a bug. In contrast, bug fingerprints contain additional small amounts of highly relevant runtime information that helps understand how the bug occurred. We show how these "fingerprints" can be used to speed up both manual and automated debugging. As a proof of concept, we present DCop, a system for collecting such runtime information about deadlocks and including it in the corresponding bug reports. The runtime overhead introduced by DCop is negligible (less than 0.17% for the Apache Web server), so it is suitable for use in production.

1 Introduction

Software debugging is time-consuming and labor-intensive, with most of the work going into understanding how the reported bug occurred in the field. For example, 70% of the concurrency bugs reported to Microsoft take days to months to fix [4]. This labor-intensive aspect makes software debugging expensive. Our work is aimed at reducing the often needless labor involved in debugging.

A classic bug report contains at best a coredump that describes the final state of the failed application. State-of-the-art automated bug reporting systems, such as Windows Error Reporting (WER) [3], produce a trimmed version of the coredump, which reduces the overhead and privacy problems of a full coredump. Most other software relies on users to provide a coredump and a description of how to manually reproduce the bug.

The absence of precise information about how the bug occurred in the field leads to incorrect diagnosis. Non-deterministic bugs are particularly hard to reproduce and diagnose, forcing developers to guess the cause of the failure based on the bug report. Since bug reports offer no runtime information about how the bug occurred, guessing often leads to incorrect fixes; e.g., one study reports that 30% of concurrency bugs are initially fixed incorrectly [8]. Ideally, developers should not have to rely so much on guessing.

There do exist tools that help reduce guessing by employing full system record-replay [2]. However, they can incur runtime recording overheads that make them impractical for in-production use. Other approaches use post-factum analysis to eliminate completely the need for runtime recording (execution synthesis [15]), or to require less recording (ODR [1], SherLog [14]). However, post-factum analysis may not be effective for all bugs and can involve substantial compute time at the developer's site.

Appears in Proceedings of the Intl. Conference on Runtime Verification, Malta, November 2010

The inherent trade-off between runtime recording overhead and the fidelity/ease of subsequently reproducing bugs forms a spectrum of solutions, with full system replay [2] at one end, and execution synthesis [15] at the other. This spectrum is still poorly understood, and an important question remains: which is the least amount of information that is practical to record at runtime, yet still makes it easy to diagnose bugs of a certain type.

Our observation is that, given a class of bugs, it is possible to record a small amount of bug-specific runtime information with negligible overhead, and this information can substantially improve debugging. Based on this observation, we propose *bug fingerprints*, small additions to classic bug reports that contain highly relevant "breadcrumbs" of the execution in which the bug occurred. These breadcrumbs ease the reconstruction of the sequence of events that led to the failure.

We show that this idea works for deadlocks, an important class of concurrency bugs. We built DCop, a prototype deadlock fingerprinting system for C/C++ software—it keeps track at runtime of each thread's lock set and the callstacks of the corresponding lock acquisitions; when a deadlock hangs the application, this information is added to the bug report. DCop's runtime overhead is negligible (e.g., less than 0.17% for the Apache Web server), yet these breadcrumbs enable faster, even automated, debugging.

In the rest of this paper we describe the design of DCop (§2), evaluate it (§3), discuss the generalization of bug fingerprints to other bug types (§4), illustrate the use of fingerprints for automated debugging (§5), review related work (§6), and conclude (§7).

2 Deadlock Fingerprints

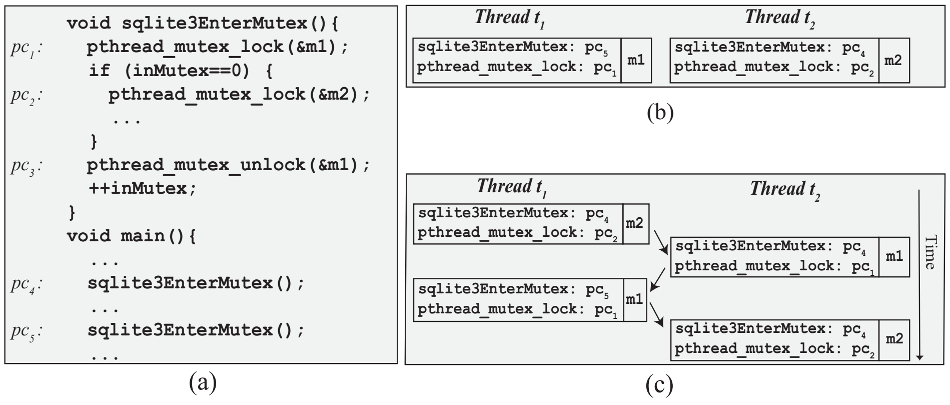
Despite being frequent (e.g., 30% of the bugs reported in [8] are deadlocks), deadlock bug reports are scarce, because deadlocks do not produce a coredump—instead, they render the application unresponsive. Normal users restart the application without submitting a bug report, while expert users may attach a debugger to the program and capture each thread's callstack. Systems such as WER can be used to create a coredump, but it is still hard to debug deadlocks based on this information that describes only the end state.

Deadlocks become straightforward to debug if we have information on how the program acquired *every* mutex involved in the deadlock. In particular, the callstacks of the calls that acquired mutexes *held* at the time of deadlock, together with the callstacks of the *blocked* mutex acquisitions, provide rich information about how the deadlock came about. Alas, the former type of callstack information is no longer available at the time of the deadlock, and so it does not appear in the coredump.

Fortunately, it is feasible to have this information in every bug report: First, the amount of information is small—typically one callstack per thread. Second, it can be maintained with low runtime overhead, because most programs use synchronization infrequently. As it turns out, even for lock-intensive programs DCop incurs negligible overhead.

DCop's deadlock fingerprints contain precisely this information. Regular deadlock bug reports contain callstacks, thread identifiers, and addresses of the mutexes that are requested—but not held—by the deadlocked threads. We call these the *inner* mutexes, corresponding to the innermost acquisition attempt in a nested locking sequence. Additionally, deadlock fingerprints contain callstack, thread id, and address information for the mutexes that are already held by the threads that deadlock. We call these the *outer* mutexes, because they correspond to the outer layers of the nested locking sequence. Outer mutex information must be collected at runtime, because the functions where the outer mutexes were acquired are likely to have already returned prior to the deadlock.

We illustrate deadlock fingerprints with the code in Fig. 1a, a simplified version of the global mutex implementation in SQLite [11], a widely used embedded database engine. The bug occurs when two threads execute *sqlite3EnterMutex()* concurrently. Fig. 1b shows the classic bug report, and Fig. 1c shows the deadlock fingerprint.



```
       void sqlite3EnterMutex(){
pc1:     pthread_mutex_lock(&m1);
         if (inMutex==0) {
pc2:       pthread_mutex_lock(&m2);
           ...
         }
pc3:     pthread_mutex_unlock(&m1);
         ++inMutex;
       }
       void main(){
         ...
pc4:     sqlite3EnterMutex();
         ...
pc5:     sqlite3EnterMutex();
         ...
```

(a)

```
Thread t1                                 Thread t2
sqlite3EnterMutex: pc5        m1          sqlite3EnterMutex: pc4        m2
pthread_mutex_lock: pc1                   pthread_mutex_lock: pc2
```

(b)

```
Thread t1                                 Thread t2
sqlite3EnterMutex: pc4        m2
pthread_mutex_lock: pc2
                                          sqlite3EnterMutex: pc4        m1
                                          pthread_mutex_lock: pc1
sqlite3EnterMutex: pc5        m1
pthread_mutex_lock: pc1
                                          sqlite3EnterMutex: pc4        m2
                                          pthread_mutex_lock: pc2
```

(c)

Fig. 1. (a) SQLite deadlock bug #1672. **(b)** Regular bug report. **(c)** DCop-style deadlock fingerprint.

A regular bug report shows the final state of the deadlocked program: t_1 attempted to lock mutex m_1 at pc_1 and t_2 attempted to lock mutex m_2 at pc_2—we invite the reader to diagnose how the deadlock occurred based on this information. The bug report does not explain how t_1 acquired m_2 and how t_2 acquired m_1, and this is not obvious, since there are several execution paths that can acquire mutexes m_1 and m_2.

The deadlock fingerprint (Fig. 1c) clarifies the sequence of events: t_1 acquired m_2 at pc_2 in a first call to *sqlite3EnterMutex*, and t_2 acquired m_1 at pc_1. This allows a developer to realize that, just after t_1 unlocked m_1 at pc_3 and before t_1 incremented the *inMutex* variable, t_2 must have locked m_1 at pc_1 and read variable *inMutex*, which still had the value 0. Thus, t_2 blocked waiting for m_2 at pc_2. Next, t_1 resumed, incremented *inMutex*, called *sqlite3EnterMutex* the second time, and tried to acquire m_1 at pc_1. Since m_1 was held by t_2 and m_2 was held by t_1, the threads deadlocked. This is an example of how DCop can help debug the deadlock and reveal the data race on *inMutex*.

To acquire this added information, DCop uses a lightweight instrumentation layer that intercepts the program's synchronization operations. It records the acquisition callstack for currently held mutexes in a per-thread event list. A deadlock detector is run whenever the application is deemed unresponsive, and it determines whether the cause is a deadlock.

The runtime monitor is designed to incur minimal overhead. First key decision was to avoid contention at all costs, so each thread records the callstack information for its lock/unlock events in a thread-local private list. The private lists are merged solely when a deadlock is found (and thus the application threads are stuck anyway). This avoids introducing any additional runtime synchronization.

A second design choice was to trim the private lists and keep them to the minimum required size: every time a mutex is unlocked, DCop finds the corresponding lock event in the list and discards it—mutexes that are no longer held cannot be involved in deadlocks. Thus, DCop only keeps track of mutexes that have not yet been released, and so the size of a per-thread event list is bounded by the maximum nesting level of locking in the program. In our experience, no event lists ever exceeded 4 elements.

As a result of this design, DCop's runtime overhead is dominated by obtaining the backtrace on each mutex acquisition. To reduce this overhead to a minimum, DCop resolves backtrace symbols offline, since this is expensive and need not be done at runtime.

The deadlock detection component of DCop is activated when the user stops an application due to it being unresponsive. The detector processes each thread's list and creates a resource allocation graph (RAG) based on the events in the lists. The RAG contains a vertex for each active thread and mutex, and edges correspond to mutex acquisitions (or acquisition requests that have not succeeded yet). Edges are labeled with the thread id of the acquiring thread and the callstack corresponding to the lock operation. Once the RAG is constructed, the detector checks for cycles in the graph—a RAG cycle corresponds to a deadlock. If a deadlock is found, the detector assembles the corresponding fingerprint based on the callstacks and thread identifiers found on the cycle's edges.

DCop's deadlock detector has zero false positives. Furthermore, since the size of the threads' event lists is small, assembling a deadlock fingerprint is fast.

We implemented DCop inside FreeBSD's `libthr` POSIX threads library; our changes added 382 LOC. One advantage of recording fingerprints from within the existing threading library is the opportunity to leverage existing data structures. For example, we added pointers to DCop's data structures inside the library's own thread metadata structure. An important optimization in DCop is the use of preallocated buffers for storing the backtrace of mutex acquisitions—this removes memory allocations from the critical path.

3 Performance Evaluation

Having discussed DCop's design, we now turn our attention to the key question of whether it is suitable for use in production? We evaluate DCop's performance on a workstation with two Intel 4×1.6GHz-core CPUs with 4GB of RAM running FreeBSD 7.0.

First, we employ DCop on interactive applications we use ourselves, such as the emacs text editor. There is no perceptible slowdown, leading to the empirical conclusion that user-perceived overhead is negligible. However, since recording mutex operations adds several instructions at each synchronization operation, (e.g., obtaining the backtrace for a lock operation), some lock intensive programs may exhibit more overhead.

Next, we use DCop for the Apache Web server with 50 worker threads. We vary the number of concurrent clients and, for each concurrency level, we execute 5×10^5 GET requests for a 44-byte file. In Fig. 2 we compare the aggregate request throughput to a baseline without DCop. The overhead introduced by DCop is negligible throughout, with the worst-case being a less than 0.17% drop in throughput for 200 concurrent clients. Both baseline and DCop throughput decrease slightly with concurrency level, most likely because there are

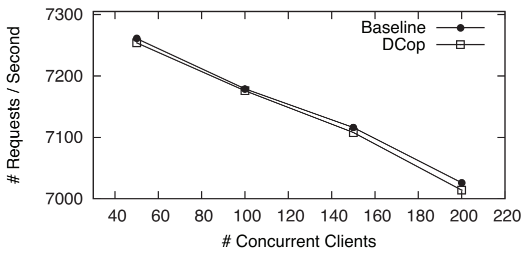


Fig. 2. Comparative request throughput for the Apache 2.2.14 server at various levels of client concurrency.

more clients than worker threads. The maximum synchronization throughput (lock operations/second) reaches 7249 locks/second.

To analyze DCop's overhead in depth, we wrote a synchronization-intensive benchmark that creates 2 to 1024 threads that synchronize on 8 shared mutexes. Each thread holds a mutex for δ_{in} time, releases it, waits for δ_{out} time, then tries to acquire another mutex. δ_{in} and δ_{out} are implemented as busy loops, thus simulating computation done inside and outside a critical section. The threads randomly call multiple functions within the microbenchmark, in order to build up highly varied callstacks ("fingerprints").

We measure how synchronization throughput varies with the number of threads. In Fig. 3 we show DCop's overhead for δ_{in}=1 microsecond and δ_{out}=1 millisecond, simulating a program that grabs a mutex, updates in-memory shared data structures, releases the mutex, and then performs computation outside the critical section. The worst case overhead is less than 0.33% overhead. The decreasing overhead shows that indeed DCop introduces no lock contention. Instead, the application's own contention amortizes DCop's overhead.

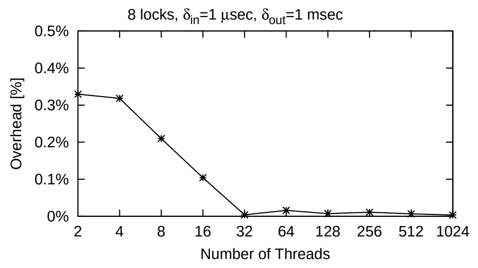


Fig. 3. Overhead of collecting deadlock fingerprints as a function of the number of threads.

We repeat the experiment for various combinations of $1 \leq \delta_{in} \leq 10^4$ and $1 \leq \delta_{out} \leq 10^4$ microseconds, simulating applications with a broad range of locking patterns. The measured overhead ranges from 0.06% in the best case to 0.77% in the worst case. The maximum measured synchronization throughput reaches 831,864 locks/second.

These results confirm that DCop introduces negligible runtime overhead, thus making it well suited for running in production, even for server applications. We hope this advantageous cost/benefit trade-off will encourage wider adoption of deadlock fingerprinting.

4 Generalizing Bug Fingerprinting

Having seen how bug fingerprinting works for deadlocks, we now turn our attention to generalizing bug fingerprinting to other kinds of bugs. In §5 we discuss how bug fingerprints can be employed in automated debugging.

In essence, a bug fingerprint serves to *disambiguate executions*: when faced with a bug report, a developer must find (guess) which of the many (often infinite) possible executions of the software could have led to the observed failure. The bug report provides clues for trimming down the set of possible executions, and the bug fingerprint should narrow it down to only a handful of possibilities. Fingerprint information must be small, to avoid undue recording overheads. Choosing what runtime information to include in a given fingerprint is therefore specific to each class of bugs. We illustrate this process with two examples: data races and unchecked function returns.

A bug fingerprint for a data race-induced failure contains information on the races that manifested during execution prior to the failure in the bug report. This way, it is possible to determine which potential data races influenced the execution and which did not. However, monitoring memory accesses efficiently is not easy.

An efficient data race fingerprinting system employs static analysis to determine offline, prior to execution, which memory accesses are potential data races. It then monitors at runtime only these accesses. There are two options to perform such monitoring with low overhead: debug registers and transactional memory (TM). x86 debug registers [6] can be configured to deliver an interrupt to a monitor thread whenever two memory accesses to the same address are not ordered by a happens-before relation and at least one of the access is a write (i.e., a data race occurred). The corresponding program counters and memory address are then saved for later inclusion in the bug report, should a failure occur. One drawback is that today's CPUs can monitor only a small set of addresses at a time, so debug registers can be used to watch only a subset of the statically-discovered potential races. An alternative approach is to use the conflict detection mechanism of TM to detect data races, and record the fingerprint. If TM features are available in hardware, this can be done quite efficiently.

Another interesting class of bugs appears in code that "forgets" to check all possible return values of a library function. For example, not checking whether a socket `read()` call returned -1 can lead to data loss (if caller continues as if all data was read) or even memory corruption (if return value is used as an index). For such unchecked-return bugs, the fingerprint contains (a) the program locations where a library function call's return value was not checked against all possible return values, and (b) the actual return value. Such fingerprinting can be done with low overhead by statically analyzing the program binary to determine the places in the program where library calls are not properly checked (e.g., using the LFI callsite analyzer [9]), and monitoring at runtime only those locations.

For most bug types, a general solution is to incrementally record the execution index [12] and include it in the bug fingerprint. The execution index is a precise way to identify a point in an execution and can be used to correlate points across multiple executions. Such a bug fingerprint can be used to reason with high accuracy about the path that the program took in production, but has typically high recording overhead (up to 42% [12]). It is possible to reduce the overhead by recording only a partial execution index (e.g., by sampling) that, although less precise, can still offer clues for debugging.

It is practical to fingerprint any class of bugs, as long as the runtime information required to disambiguate possible executions that manifest the bug can be recorded efficiently. Fingerprinting mechanisms can leverage each other, so that collecting fingerprints for *n* classes of bugs at the same time is cheaper than *n* times the average individual cost.

5 Debugging Using Deadlock Fingerprints

Augmenting bug reports with bug fingerprints can substantially speed up debugging. For example, a developer debugging a deadlock can get from the deadlock fingerprint all mutexes involved in the deadlock and the callstacks corresponding to their acquisition calls. This allows the developer to insert breakpoints at all outer mutex locations and understand how the deadlock can occur.

Bug fingerprints are also an excellent aid for automated debuggers, like ESD [15]. ESD is based on execution synthesis, an automated technique that starts from a bug report and finds an execution path and a thread schedule that reproduce the bug deterministically, with no human intervention. The technique employs a static analysis phase, that proceeds backward from the coredump and identifies critical transitions that take the program to the state contained in the coredump. Then a forward symbolic execution phase searches for the necessary inputs and thread schedule to reproduce the bug.

Bug signatures can improve the efficiency of execution synthesis, since they help disambiguate between possible executions. The more execution paths appear to be likely to reach the end state contained in the coredump, the longer ESD has to search. Bug signatures, however, contain clues that can substantially prune this search space.

For example, a major challenge in execution synthesis for deadlocks is identifying the thread schedule that leads to deadlock. DCop's deadlock fingerprints narrow down the set of possible schedules, thus reducing search time. In preliminary measurements, we find that for a program with three threads and an average lock nesting level of three, the thread schedule synthesis phase of ESD can be reduced by an order of magnitude. Similarly, in the case of data races, we expect orders of magnitude improvement in search performance, if data race fingerprints are available.

The combination of low-overhead bug fingerprinting with ESD-style automated debugging promises to improve the productivity of software developers. Thus, we consider a combined deployment to be an appealing solution for the software industry.

6 Related Work

Runtime support for debugging ranges from classic bug reporting systems, that provide a partial or complete coredump, to heavyweight whole-system record-replay systems. Special hardware can be used to make the latter approach faster. In between these extremes, there exist multiple approaches that record less information and use post-factum analysis to reconstruct the missing pieces offline. We briefly survey this spectrum of solutions.

The state of the art in automated bug reporting systems, such as Windows Error Reporting [3], collect bug reports from a large number of users. These bug reports reveal some information about the bug (e.g., the end state of the application), but not how the application got there. Bug fingerprints enrich bug reports with bug-specific runtime information that can help these systems classify failures more accurately. As shown earlier, bug fingerprints preserve the low runtime overhead of classic bug reporting systems.

FDR [13] uses modified hardware to perform efficient execution recording. It piggybacks on the cache coherence hardware to record thread ordering information. While this approach can help debugging, it requires hardware features that are not available today and that are uncertain to exist in the future. In contrast, a system like DCop can be used today, without requiring any hardware or software changes.

Other approaches record system execution at the virtual machine level and use this information to deterministically replay executions. They are highly precise, but can incur significant overhead (e.g., up to 260% for Revirt [2]); recording multiprocessor executions has typically several orders of magnitude higher overhead. Bug fingerprints operate at a higher level: they leverage knowledge about the bug class to identify minute pieces of runtime information that help reproduce the bugs with minimal recording. Although they require more human effort and they lack the precision of VM-based record-replay, bug fingerprints are an effective debugging aid with virtually no runtime overhead.

R2 [5] performs record-replay at the library level, and can interpose at high-level APIs to reduce the recording overhead. R2 offers the flexibility of choosing what exactly to record, so it is in essence a mechanism for performing selective recording. We believe R2 could be used to obtain fingerprints for certain classes of bugs, although R2 has limited support for nondeterministic executions. That being said, DCop incurs two orders of magnitude less overhead than R2 on Apache, for identical workloads.

ODR [1] and PRES [10] are recent systems for replaying concurrency bugs; they trade runtime overhead for post-factum analysis time, and thus explore new points in the spectrum of solutions. The benefit of deterministic replay comes at a cost of more than 50% runtime overhead, which makes them less compelling for production use. In DCop we forgo the goal of deterministic replay in exchange for negligible runtime overhead.

Dimmunix [7] also collects deadlock fingerprints, but for a different reason: immunity against deadlocks. Once a deadlock occurs, Dimmunix records a signature of the deadlock that is then used to identify and avoid that same deadlock pattern in subsequent executions. Since DCop is focused on collecting fingerprints, not on avoidance, it can perform the collection with two orders of magnitude less runtime overhead and produce fingerprints that are richer than Dimmunix's signatures.

7 Conclusions

This paper described *bug fingerprints*, an augmentation of classic bug reports with runtime information about how the reported bug occurred. Fingerprints contain clues that substantially help in both manual and automated debugging. A proof-of-concept system fingerprints deadlocks with negligible overhead (less than 0.17% for Apache). We discussed how to extend this approach to other types of bugs, and argued that coupling bug fingerprints with automated debugging techniques can make debugging more efficient.

References

1. G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore programs. In *Symp. on Operating Systems Principles*, 2009.
2. G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. on Operating Systems Design and Implementation*, 2002.
3. K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.
4. P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.
5. 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 *Symp. on Operating Systems Design and Implementation*, 2008.
6. Intel. *Intel Architecture Software Developer's Manual, Vol. 2: Instruction Set Reference*. 1999.
7. H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Symp. on Operating Systems Design and Implementation*, 2008.
8. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.
9. P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conf.*, 2010.
10. S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. Do you have to reproduce the bug at the first replay attempt? – PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles*, 2009.
11. SQLite. http://www.sqlite.org/, 2010.
12. B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Conf. on Programming Language Design and Implementation*, 2008.
13. M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Intl. Symp. on Computer Architecture*, 2003.
14. D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.
15. C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.