Light64: Lightweight Hardware Support for Data Race Detection during Systematic Testing of Parallel Programs

Adrian Nistor
University of Illinois at Urbana-Champaign, USA
nistor1@illinois.edu

Darko Marinov
University of Illinois at Urbana-Champaign, USA
marinov@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

Developing and testing parallel code is hard. Even for one given input, a parallel program can have many possible different thread interleavings, which are hard for the programmer to foresee and for a testing tool to cover using stress or random testing. For this reason, a recent trend is to use Systematic Testing, which methodically explores different thread interleavings, while checking for various bugs. Data races are common bugs but, unfortunately, checking for races is often skipped in systematic testers because it introduces substantial runtime overhead if done purely in software. Recently, several techniques for race detection in hardware have been proposed, but they still require significant hardware support.

This paper presents *Light64*, a novel technique for data race detection during systematic testing that has both small runtime overhead and very lightweight hardware requirements. Light64 is based on the observation that two thread interleavings in which racing accesses are flipped will very likely exhibit some deviation in their program execution history. Light64 computes a 64-bit hash of the program execution history during systematic testing. If the hashes of two interleavings with the same happens-before graph differ, then a race has occurred. Light64 only needs a 64-bit register per core, a drastic improvement over previous hardware schemes. In addition, our experiments on SPLASH-2 applications show that Light64 has no false positives, detects 96% of races, and induces only a small slowdown for race-free executions — on average, 1% and 37% in two different modes.

Categories and Subject Descriptors

B.3.4 [**Hardware**]: Reliability, Testing, and Fault-Tolerance—*Error-checking*; D.1.3 [**Concurrent Programming**]: Parallel programming.

General Terms

Reliability, Verification.

Keywords

Systematic Testing, Data Race, Execution History Hash.

MICRO'09, December 12–16, 2009, New York, NY, USA.

1. INTRODUCTION

Multicore hardware brings the challenge of parallel programming to the large body of programmers. Even when parallel programs are written by expert programmers and acknowledged as safety critical, concurrency bugs such as data races have been known to cause serious accidents [5, 20]. Consequently, it is likely that, as the number of parallel programmers increases, concurrency bugs will become a source of great concern for the software industry.

A fundamental tool to combat bugs is software testing. Unfortunately, testing parallel code is intrinsically hard because, often, for the same input, there are many possible different thread interleavings. These interleavings can be difficult for the programmer to foresee and for a testing tool to cover using stress or random testing [2, 26]. For this reason, *Systematic Testing*, where a tool systematically explores the interleaving space of a parallel program while checking for bugs, is emerging as an effective approach for parallel and distributed program testing [3,4,14,15,23,29–34]. Systematic testing is based on ideas from state-space exploration and model checking. It typically assumes that test inputs are provided, and that the only source of non-determinism is due to thread interleavings. For a given input, systematic testing executes the program many times, each time forcing a different thread interleaving. When compared to stress testing or heuristic approaches [2,19,26], systematic testing can offer higher coverage guarantees on what has been tested. A highly effective systematic tester used in industry is CHESS from Microsoft Research [15]. In one comparison to stress testing [15] on large code bases (e.g., operating system and distributed execution engine), CHESS found and was able to reproduce 25 previously unknown bugs that stress testing did not find or could not reproduce for many months.

Data races are a common type of concurrency bug, and thus systematic testers commonly include functionality for data race detection [3, 15, 29, 30]. They usually employ standard detection algorithms, such as Eraser [25], which do not take advantage of the systematic testing environment and have a high runtime overhead. For example, race detection in CHESS currently has about 30x overhead [1]. For this reason, systematic testers do not always enable race detection — e.g., CHESS turns it off by default [15] — which may result in missing not only the data races themselves but also the exploration of new states induced by the races. To reduce overheads to manageable levels, several recent projects have proposed hardware support for race detection [13, 16, 21, 22, 35]. Unfortunately, such schemes still require significant hardware extensions.

To address this problem, this paper presents *Light64*, a new technique for efficient detection of data races during systematic testing that has both small runtime overhead and very lightweight hardware requirements. Light64 is based on the following observation: two different thread interleavings that have the same *happens-*

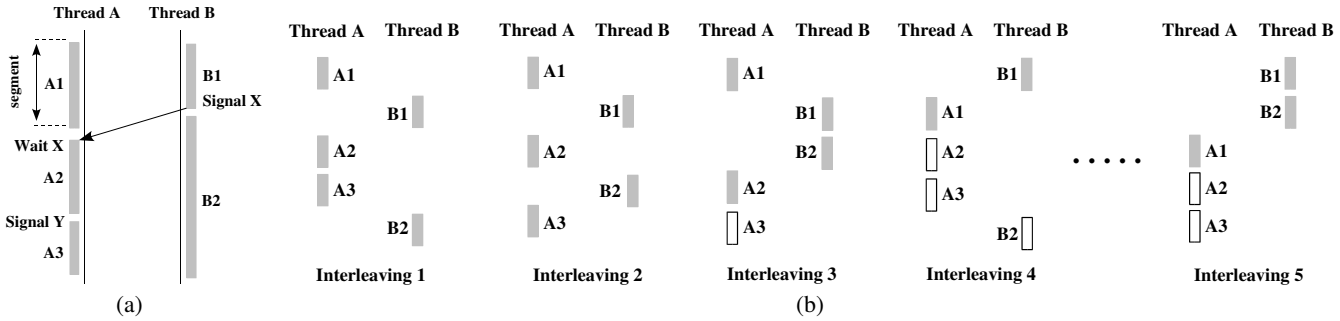


Figure 1: Parallel execution seen as a succession of segments (a), and some of the segment interleavings generated in Depth First Search (DFS) order (b).

before graph [8, 12] but a flipped (i.e., reordered) data race, will very likely have at least a small deviation in the execution history of some thread. Consequently, Light64 collects the execution histories of the two interleavings and, if they differ, it reckons that a data race is present. Then, it saves an execution log, from which an offline classical data race analysis [12] precisely detects the racing data accesses.

This paper makes the following contributions:

Novel technique with low hardware requirements and low execution overhead: Light64 is a novel technique for detecting data races during systematic testing of multithreaded programs. Light64 has a low execution overhead because it performs a very fast online analysis to detect that a race *did occur* (which is infrequent) followed by a slower offline analysis to determine *where* the race occurred. We also present an optimization that reduces the number of logs for offline analysis by orders of magnitude by detecting that some logs are due to the same race.

Light64's online analysis compares execution histories, relying on the re-execution of program sections and a highly deterministic environment, which are typically provided by systematic testers. For fast comparison, Light64 efficiently encodes execution histories. Specifically, Light64 hashes the values read by loads (rather than addresses or any other part of the execution history), which is enough to detect the changes due to data races. If this hashing were done purely in software, the overhead could be high. Our hardware scheme has the extremely lightweight hardware requirement of only a 64-bit hash register per core — a drastic improvement over all other hardware techniques for detecting data races [13, 16, 21,22,35]. In addition, Light64 trivially supports *virtualization* and process *migration*, which is important for deployment in real-world systems. Most other hardware techniques [13, 21, 22, 35] do not allow for such operations, while the only technique that does [16] adds hardware for this purpose.

Effective integration into systematic testing: The key insight of Light64 is to detect differences in execution history for interleavings that have the same happens-before graph. Light64 can rely on systematic testers to produce executions with the same happens-before and different thread interleavings. We call this Light64 mode *Passive*. We also introduce a new mode, called *Active*, which intentionally reshuffles interleavings during exploration to increase the chance to flip data races at the cost of a slightly higher runtime overhead. We describe several variations of this mode which trade off capability of race detection for execution overhead.

Effective Data Race Detection: We evaluate Light64 on all the programs from the SPLASH-2 benchmark suite. The results show that Light64 has no false positives, detects 96% of races, and induces a race-free runtime overhead of only 1% and 37% on average — for Passive and Active modes, respectively. This overhead is much smaller than that of existing software schemes, and comparable to existing hardware schemes that require significantly more hardware extensions. Moreover, Light64 can detect races separated by arbitrary distances in execution time, while all other hardware techniques for race detection lose races between accesses that are far apart in the execution due to limited hardware storage capabilities [16, 21, 22, 35].

The outline of the paper is as follows. Section 2 gives a high-level overview of systematic testing. Section 3 presents Light64. Section 4 describes the hardware support and the software algorithms. Section 5 gives the experimental results. Section 6 reviews related work, and Section 7 concludes the paper.

2. BACKGROUND

2.1 Systematic Testing of Parallel Programs

Systematic testing of a parallel program [3, 4, 14, 15, 23, 29–34] involves methodically exploring different thread interleavings of the program (for a given input), executing each interleaving, and checking for bugs during these executions. To test a given interleaving, systematic testing *multiplexes* the threads of the program on a single processor. However, different processors can be assigned to test different interleavings at the same time [6]. The bugs being checked for can include data races, deadlocks, assertion violations, memory leaks, etc. The checks are usually independent of the exploration and are implemented using standard algorithms that run on top of the systematic tester.

To control the execution of the program being checked, a systematic tester implements its own thread scheduler, which replaces the scheduler from the operating system in case of C/C++ programs [4, 14, 15, 31, 34] or from the virtual machine in case of Java [3, 29, 30]. Systematic testers do not explore thread interleavings at the instruction granularity but at the *segment* granularity. A segment is a dynamic section of code that contains a single communication operation (e.g., a synchronization operation, an access to a shared variable, or a message). During testing, systematic testers run each segment atomically, and repeatedly try different legal segment orders to explore different interleavings (for a given input). Exploring all communication interleavings covers the same program behavior as exploring all instruction interleavings, but at a much lower cost.

For example, Figure 1(a) shows a simplified program fragment with two threads, A and B, that have three and two segments, respectively. The only communication in this example is the *Signal/Wait* synchronization; besides it, the segments are *independent*, i.e., they do not communicate. Figure 1(b) shows some of the

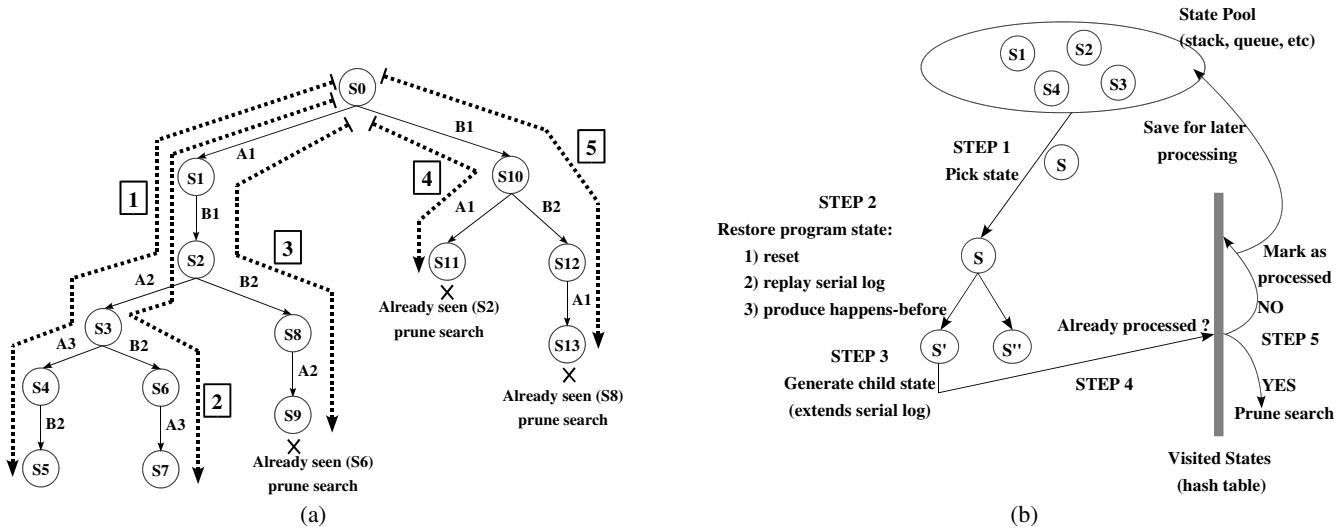


Figure 2: The State Tree with a DFS exploration (a), and the operation of a systematic tester (b).

thread interleavings (at segment granularity) that a systematic tester explores using the Depth First Search (DFS) strategy. The white segments in interleavings 3, 4, and 5 are not actually executed but rather pruned away, as we will describe later in this section.

Figure 2(a) shows the same thread interleavings as a *State Tree*. Each node represents the *State* of the parallel program between two thread switches. Each edge represents the atomic execution of a segment. For example, the edge between states S1 and S2 corresponds to the execution of segment B1 from Figure 1(a). Figure 2(a) is augmented with dotted lines showing each of the interleavings of Figure 1(b).

The number of children of a state is equal to the number of *enabled threads* that can be executed from that state. For example, state S1 has only one child, corresponding to the execution of thread B, since thread A is blocked at the *Wait X* operation. State S4 has only one child because thread A has finished execution.

Two states are *equivalent* if exploring the subtree from one state yields the same behavior information as exploring the subtree from the other state. Systematic testers use this property to *prune* the exploration of the state tree. In Figure 1(b), the white segments correspond to the execution that was pruned. For example, the state S9 in Figure 2(a) (which is reached after executing A2 in interleaving 3 of Figure 1(b)) has already been seen as S6 in Figure 2(a) (which is reached after executing B2 in interleaving 2 of Figure 1(b)). Consequently, the exploration beyond S9 is pruned. Overall, by traversing this pruned state tree, the systematic tester explores all the different program behavior from the interleaving point of view for a given input. Note that systematic testers build the tree on the fly, while it is being explored.

Figure 2(b) depicts the operation of a systematic tester. The systematic tester keeps in a *State Pool* the set of states that need to be processed. Each state is represented with a *Serial Log*, which is the ordered list of the segments that are executed to reach the state. For example, state S3 is represented by the Serial Log (A1-B1-A2). The State Pool is a stack for DFS or a queue for Breadth First Search (BFS). Initially, it only has the starting state of the program.

The systematic tester also keeps a *Visited States* table, with the set of states already processed during exploration. A state can be represented with its happens-before graph, which is composed of the unordered list of the segments that were executed to reach the state, plus the set of inter-thread communications that took place [15]. For example, the happens-before for states S6 and S9 is (A1, B1, A2, B2, B1->A2). The Visited States is usually implemented as a hash table. Initially, it is empty.

The main loop of the systematic tester is as follows (Figure 2(b)). In step 1, the tester picks a state S to process next — removing it or not from the State Pool depending on the search strategy. The tester then restores the program state to that state (step 2), for example by resetting the program and re-executing it following S's Serial Log. At the same time, the tester obtains the happens-before graph of the state, either by reading it from some location, or by computing it on the fly. Then, for each of the enabled threads, the tester will perform three more steps (only shown for one enabled thread in Figure 2(b)). Specifically, in step 3 it executes the next segment of the enabled thread, therefore generating a child state S'. In step 4, it checks in the Visited States if the tester has already processed a state with the same happens-before as the child state. If so, the child is discarded, effectively pruning the search; otherwise, the child state is added to Visited States and State Pool (step 5).[1]

In Figure 2(a), the order in which states are visited in DFS is obtained by first following the dotted line labeled 1. Then, the program is returned to state S3 and proceeds to follow dotted line 2 to state S7; then, the program is returned to state S2 and proceeds to follow dotted line 3 to state S9, before moving to dotted lines 4 and 5. As we return the program to a previous state, we end up re-executing some sequences of program segments. For instance, to return to state S3, the tester resets the program and re-executes A1, B1, and A2 in step 2 of Figure 2(b).

The tester prunes the exploration at states S9, S11, and S13 because they are equivalent to states S6, S2, and S8, respectively. For example, state S11 is equivalent to state S2 because both have the same happens-before graph — both are reached by executing independent segments A1 and B1 in a different serial order. This corresponds to a hit in Visited States in step 4 of Figure 2(b).

While we have explained step 2 of Figure 2(b) using a reset and replay log approach, it can also be implemented using a checkpoint-restore approach. Most C/C++ systematic testers [4, 15, 31, 34] use

[1]Some systematic testers do not maintain Visited States but rather compute on the fly equivalent states, at the cost of failing to detect some equivalent states.

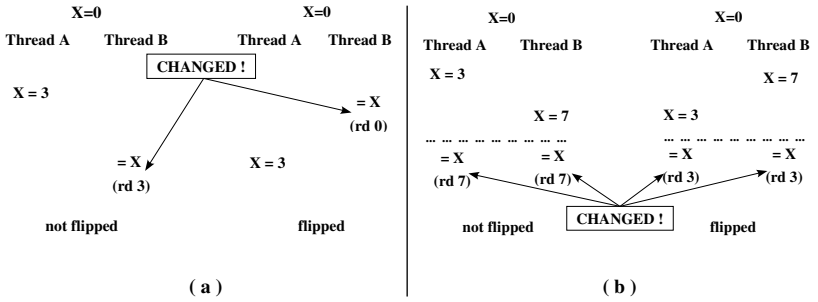


Figure 3: Examples of wr-rd (a) and wr-wr (b) data races.

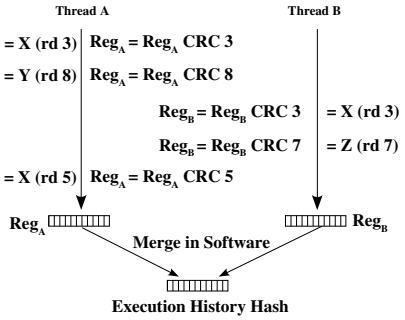


Figure 4: Computing the Execution History Hash.

reset and replay. One exception is CMC [14], but it is specialized for checking communication protocols and does not capture the entire machine state, including the stack and the registers. Java systematic testers usually use the checkpoint-restore approach [3, 30], although sometimes they use reset and replay [29]. Light64 works with both approaches.

Finally, in a systematic testing environment, the tester restricts all sources of non-determinism beyond thread execution interleaving that can affect a program's execution [4, 14, 15, 30] — such as non-deterministic system calls like gettimeofday or calls to random number generators. This is because the tester needs to be in full control of the state exploration.

2.2 Happens-Before and Data Races

The ordering between two accesses to the same memory location introduces a *happens-before edge* [8]. In Figure 1(a), the happens-before edge between synchronizations (i.e., *synchronization happens-before*) is marked with an arrow. There can also be happens-before edges between data accesses (i.e., *data happens-before*). All happens-before edges in an execution form a *happens-before graph*, which fully characterizes the ordering in a parallel execution [8].

Two data accesses to the same location are in a race if at least one of them is a write and they are not ordered by synchronization [28]. In the absence of data races, the synchronization happens-before implies the data happens-before — i.e., two data accesses to the same location are ordered by synchronization happens-before. Therefore, the latter fully characterizes the execution. In the rest of the paper, we refer to the synchronization happens-before simply as the *happens-before*.

Practical systematic testers [3, 15] determine execution segments only based on synchronization operations (and not accesses to shared variables), which makes the segments longer, their number smaller, and thus results in fewer interleavings. In effect, they assume that there is no data race. If there is one, they not only miss the race but also some possible interleavings/states resulting from the race.

3. LIGHT64

3.1 The Idea

Light64 is based on the observation that, if we flip the order of two racing accesses, we are very likely to cause a deviation in the program execution history. This can be seen in Figure 3(a), which shows a program with Threads A and B racing to access variable X. Initially, X had value 0. In the left part of Figure 3(a) (non-flipped), Thread A writes value 3 to X, and then Thread B reads X. In the right part of Figure 3(a) (flipped), Thread B reads X and then Thread A writes 3. In the example, the flipping creates a change in the execution history, namely that Thread B reads either 3 or 0. Light64 focuses on detecting such small changes, irrespective of whether or not, later in the program, this small change results in a big deviation in the program execution.

Based on this discussion, Light64 searches for two different thread execution interleavings that have (1) the *same* happens-before graph and (2) a *different* memory access history for some thread. For efficiency, Light64 tracks only a small part of the history — yet enough to detect that a deviation in execution has occurred. Specifically, Light64 computes a per-thread *hash* of the data values that each thread reads with load instructions. If the per-thread hashes of two different thread interleavings that have the same happens-before graph differ, then these interleavings have flipped a data race. Then, in an offline phase, Light64, localizes the racing instructions using a classical precise happens-before analysis [12].

Accumulating the hashes of the values read from memory collects only a small subset of the entire execution history of the program. However, it is a subset that is very useful to detect when a race has been flipped. Indeed, we will see that Light64 has *no false positives*. This means that the execution logs that Light64 saves for offline analysis will always contain a data race. Moreover, if two hashes are identical, we will see that it is very unlikely that the execution includes a harmful race. Consequently, Light64 has *very few false negatives*.

From this discussion, it is clear that Light64 needs *two executions* of the same happens-before graph to observe a data race. Fortunately, for the most part, such two executions already take place naturally in systematic testers during the process of exploration. As a result, we will see that Light64 can be well positioned in two different steps of the operation of the systematic tester shown in Figure 2(b). The resulting two designs give rise to two Light64 modes of operation: Passive and Active.

3.2 Detecting Execution History Deviations

The goal of Light64 is to efficiently detect deviations in the execution history of a program caused by the flipping of a race. To capture the execution history, we use hardware that automatically hashes and accumulates per-thread execution information.

A naive design would include in the hash every single instruction executed by the thread. However, since the inputs to the program are given by its load instructions, it is enough to hash only load instructions. Moreover, of all the parts of a load instruction — e.g., instruction address, data address, or data value —- only the data value needs to be accumulated in the hash. This is because the *very first* deviation in the execution caused by a race will involve reading a different data value from a correct address. Later, reads may use this data as an address and read from different addresses than in the

original execution. However, since detecting even a *single* deviated instruction is enough to expose the race, Light64 only hashes and accumulates the values read by load instructions.

Figure 4 shows the operation. As shown in the example, each thread uses CRC to hash and accumulate in hardware into a local register all the values read. When we reach the destination state, the hash registers of all the threads are combined in software — e.g., using another hashing function. We call the result the *Execution History Hash*.

Using this approach, if the Execution History Hash of two executions that have the same happens-before graph differ, it is due to a data race. This is because, in the highly deterministic environment of a systematic tester (Section 2.1), the hashes cannot differ for any other reason. Therefore, Light64 has *no false positives*. The execution logs that Light64 saves for offline analysis will always contain a data race. Figure 3(a) shows an example where two executions have different Execution History Hashes because of flipping a wr-rd race.

It is very unlikely that two executions with the same happens-before graph flip a harmful race and the Execution History Hash remains the same. This occurs when the hashing is such that two different streams of read values produce the same final hash value. In this case, Light64 misses the race and, therefore, we have a false negative. Since we use a 64-bit CRC, this case is extremely improbable.

Note that Light64 can miss a special case of *benign* race where the racing write writes the value that already exists. For example, in Figure 3(a), if Thread A writes 0 to X rather than 3, then Light64 misses the race. However, such race does not affect execution.

In the case of a wr-wr race (Figure 3(b)), the read that changes is the first read that *follows* the race in any thread, although the read itself may or may not be involved in a race (e.g., the dashed line in the figure could indicate a synchronization operation such as a barrier). Consequently, Light64 would miss the special case of a wr-wr race where there is no future read in the entire program execution. In our experiments (Section 5.3), Light64 *never* lost a race due to this case, and we believe that implementing more complex techniques to cover this corner case may not be justified. However, if needed, a solution would be to add a read immediately before each write — either in software by the compiler or in hardware by the microarchitecture. Since, in our environment, a segment runs between two synchronization points, one thread would perform both the read and the write before the other did. As a result, this solution would ensure that we detect when the race is flipped.

Finally, there is a special case where Light64 misses a race by construction. This case results from the fact that Light64 regards the code between two synchronization points as a segment. Since Light64 explores interleavings at the segment granularity, instructions from two different segments cannot be interleaved. As a result, if a segment operates on a variable and finally sets it to the value it had before the segment started, *and* there is a race on *that* variable, then the race is not detected. An example is shown in Figure 5(a). In the figure, variable X is initially 0. A segment in Thread A sets X to 3 and then to 0; a segment in Thread B not synchronized with A's segment reads X. Reordering the interleaving at segment granularity as in Figure 5(b) would not cause a change in the Execution History Hash. To detect this case, one would need to reorder at the instruction granularity as in Figure 5(c). Overall, while Light64 cannot detect this particular case, it is likely that, due to the high coverage of the systematic tester, the *static* race that created this *dynamic* race will be detected in other thread interleavings — namely, those where the last write of the segment does not write the same value as was at the start of it.

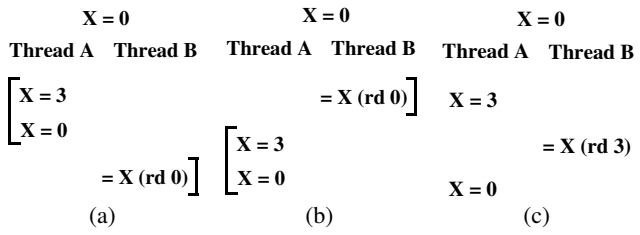


Figure 5: Reordering granularity: initial execution (a), reordering at segment granularity, (b) and reordering at instruction granularity (c).

3.3 Modes of Operation

Light64 needs two executions of the same happens-before graph to flip a race and, therefore, detect its presence. In reality, for the most part, such two executions already take place naturally in systematic testers during the process of exploring. Therefore, we place Light64 in a systematic tester so that it can take advantage of such re-executions mostly transparently.

Light64 can be positioned in two different steps of the operation of the systematic tester shown in Figure 2(b). One option is to place it in step 4. Step 4 checks the Visited States to see whether a state with the same happens-before graph as the current child state has already been processed during exploration. In this option, Light64 augments each entry in the Visited States with the Execution History Hash of the state. Then, without perturbing the exploration in any way, every time that the Visited States is accessed in step 4, Light64 simply checks both the happens-before graph and the Execution History Hash. If it finds an entry for which the former is the same and the latter differs, we have detected a data race. We call this option the Light64 Passive mode. In our experiments, this mode leads to an average runtime overhead of only 1% over plain systematic testing.

A second option is to place Light64 in step 2 of Figure 2(b). Step 2 restores a program state by using the Serial Log to replay the execution of the program up to that state. At the same time, it produces the happens-before graph of the state. In this option, Light 64 forces the re-execution to follow a *different* thread interleaving from the one it originally followed, so that it flips the races — while still obeying the same happens before graph. When the state is reached, the Execution History Hashes attained in the first execution and in this re-execution are compared. If they differ, we have detected a data race. In this option, each state in the State Pool of Figure 2(b) keeps both its happens-before graph and its Execution History Hash. The former is used to guide the re-execution in lieu of the Serial Log; the latter is used to compare against the Execution History Hash of the re-execution. We call this option the Light64 Active mode. This mode has more runtime overhead, in part because we use the happens-before graph to guide the re-execution rather than the Serial Log. In our experiments, this mode leads to an average runtime overhead of 37% over plain systematic testing. Note that this mode works only for systematic testers based on reset and replay, which is the typical choice for C/C++ systematic testers.

The Light64 Active mode keeps the overhead low by *reusing* the normal re-executions needed to restore states in systematic testing. It uses these re-executions to reorder segment execution and, therefore, to flip races. Thanks to a heuristic that we will present in the next section, one single re-execution is enough to flip typically all the races that exist in the execution path to a state. We will

present several versions of Light64 Active, which differ in whether or not additional re-executions are needed and, if so, in which special cases.

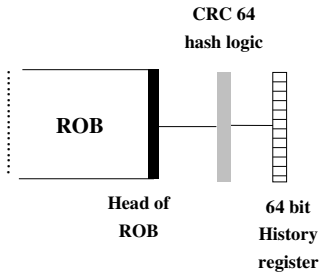
Overall, the trade-off between Active and Passive modes is that of detection capability versus overhead. Active detects races with a higher probability because it fully controls segment reordering for each happens-before graph during the exploration. Passive does not explicitly control the reordering or the re-execution of the same happens-before graph. Consequently, it may miss some races when the systematic tester does not re-execute (or re-executes but does not reorder) some thread interleaving that has a race.

4. SYSTEM DESIGN

This section describes two aspects of Light64's design, namely the hardware support and the detailed operation of the Passive and Active modes.

4.1 Hardware Support

Light64 relies on minimal hardware to generate the hash of the read history described in Section 3.2. Specifically, we place a hardware module for hash and accumulate at the head of the processor's reorder buffer (Figure 6(a)). As a load commits, the data read is automatically hashed using CRC hash logic and accumulated into the 64-bit *History* register. By hashing at instruction-commit time, we ensure a repeatable order of accesses, eliminate any perturbation from speculative accesses, and operate off the processor critical paths.



(a)

Instruction	Description
start_hashing	Start hashing the values of the memory reads
stop_hashing	Stop hashing the values of the memory reads
save_hash *addr*	Save the History register to memory location *addr*
restore_hash *addr*	Restore the History register from memory location *addr*

(b)

Figure 6: Hardware system and its interface to software.

This hardware interfaces to the software through the four assembly instructions of Figure 6(b), namely *start_hashing*, *stop_hashing*, *save_hash*, and *restore_hash*. The first two enable and disable the hashing. They allow for the analysis of only the checked program, and not also of the systematic tester's code that runs together with the checked program. The last two instructions save and restore the History register to and from memory. Recall that the hashes are computed on a per-thread basis. Therefore, Light64 uses *save_hash* and *restore_hash* to save and recover the hash at thread context switches.

With this support, we see that the Light64 hardware trivially supports virtualization and process migration. There may be multiple systematic tester programs running on the same processor, and a systematic tester program may migrate between processors if it is running on multiprocessor hardware. Light64 works seamlessly because the OS or VM monitor saves and restores a thread's 64-bit History register at thread switching points. There is no additional overhead for virtualization and migration.

4.2 Light64 Passive Mode

This mode makes use of the fact that, during a systematic tester's normal exploration process, the tester typically encounters multiple execution paths with the same happens-before graph and different segment execution orders. As an example, consider Figure 2(a). The tester reaches state S6 through the execution of A1, B1, A2 and B2. At that point, the state's happens-before graph and Execution History Hash are saved in the Visited States. Later, the tester reaches state S9, which has the same happens-before graph as S6 but was reached by executing the segments in a different order, namely A1, B1, B2, A2. In this case, A2 has been reordered relative to B2. Other paths may reorder A1 relative to B1 and B2 (note that the executions leading to S6 and S9 do not reorder A1 and B1). Specifically, the paths to S8 and S13 have the same happens-before graph and reorder A1 relative to B1 and B2. Note, however, that Light64 Passive cannot make guarantees of coverage. Due to the systematic tester's exploration strategy or pruning heuristics, two particular segments may never be reordered or may be reordered as part of thread interleavings with different happens-before graphs.

There are two infrequent corner cases in Light64 Passive. The first one is when, due to hash-induced aliasing, two different happens-before graphs are mapped to the same location of Visited States. In this case, Light64 will find different Execution History Hashes and assume that there is a race. However, the user will not receive false warnings because the offline analysis uses a precise happens-before analysis.

The second case is if a race flip affects the control flow *and* changes the happens-before graph. In this case, Light64 will be searching the wrong entry in Visited States and, therefore, may be unable to flag the presence of a data race. In this case, we would miss the race.

4.3 Light64 Active Mode

This mode reuses the normal re-executions needed to restore states in reset-and-replay systematic testers. Recall that these are the testers typically used for C/C++ [4, 15, 31, 34]. Light64 Active uses these re-executions to explicitly reorder segment execution and, therefore, flip races. This section describes the operation of Light64 Active, a heuristic for efficient segment reordering, several Light64 Active versions that trade-off accuracy and overhead, and an optimization to reduce the number of offline analysis.

4.3.1 General Operation

To illustrate the operation of the Light64 Active mode, consider Figure 2(a) at the point when S5 has been processed. Execution now needs to return to state S3 and proceed to explore S3's second child S6. In step 2 of Figure 2(b), we need to restore the program state by re-executing the path to state S3. In this re-execution, a conventional systematic tester would typically execute the segments in the same order as in the first execution, namely the order given by the Serial Log (A1-B1-A2). Light64 Active, instead, re-executes the path to state S3 re-ordering the execution of segments relative to the first execution as much as possible, while obeying the happens-before graph of S3. Specifically, it executes the segments (B1-A1-A2) — therefore flipping all the races between A1 and B1. It does not reorder B1 and A2 because they have a happens-before dependence (Figure 1(a)).

A flipped race may alter the control flow, which in turn may alter the happens-before graph. This may cause the replay to block, if a thread is waiting for a condition that never becomes true. Light64 detects this problem by keeping track of threads ready to run. If there are none and the re-execution is not yet finished, Light64 signals a race, without needing to compare the hashes; it is obvious that there was a deviation in the execution history. Systematic testers like CHESS [15] use the same approach to detect that a replay is not following the initial execution.

4.3.2 Heuristic for Efficient Segment Reordering

When replaying an execution path in Light64 Active, we want to minimize the overhead of computing and enforcing segment execution orders that flip as many segments in the path as possible. Consequently, we propose a heuristic algorithm that reorders the execution of many segments while incurring minor overhead. With two executions, this algorithm is able to flip typically all of the races in the path.

The idea is to perform one execution of the path selecting the segments to execute using the *Smallest-ID Thread First* (SID) algorithm. Then, the second execution of the path is performed while selecting the segments to execute using the *Biggest-ID Thread First* (BID) algorithm. The SID and BID algorithms order the segments of all the threads in a parallel program in a total order. At any point in the ordering, if there is more than one thread that is ready to execute a segment, the SID algorithm picks the thread with the smallest ID that is ready, while the BID algorithm picks the one with the biggest ID that is ready. Depending on the type of state exploration performed by the systematic tester (e.g., depth-first or breadth-first), this heuristic may influence the number of path re-executions. This is discussed in Section 4.3.3.

Figure 7 illustrates the SID and BID algorithms. Assume that we have two threads with IDs 1 and 2 that execute the program with the happens-before graph shown in Figure 7(a). Figures 7(b) and 7(c) show the segment execution orders when following the SID and BID algorithms, respectively.

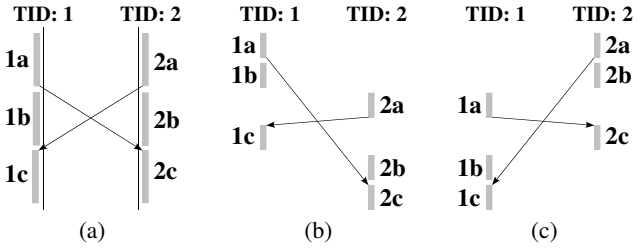


Figure 7: Heuristic to reorder segment execution. The segments of a happens-before graph (a) are executed using the SID algorithm (b) or the BID algorithm (c).

When comparing the two executions, we see that, in this example, all the segments that are not ordered by happens-before dependences get reordered. For example, in Figure 7(b), segment 1a executes before 2a and 2b, while in Figure 7(c), it executes after both of them. Consequently, any race in these segments gets flipped. On the other hand, 1a is always executed before 2c because there is a happens-before dependence between them.

When dealing with more than two threads, this heuristic may miss some legal segment reorderings. This is due to the well-known priority inversion problem [9], which arises when a high-priority task is blocked waiting for a resource held by a low-priority task; in the meantime, a medium-priority task is scheduled, thus preventing the low-priority one from executing and releasing the high-priority one. This is shown in Figure 8. The happens-before graph of Figure 8(a) is executed with the SID (Figure 8(b)) and BID (Figure 8(c)) algorithms. Segments 2a and 3b do not get reordered because, in Figure 8(c), while 3b is waiting for 1a to execute, 2a gets executed before 1a. To solve this problem, we can use existing partial solutions for priority inversion [27] or other heuristics.

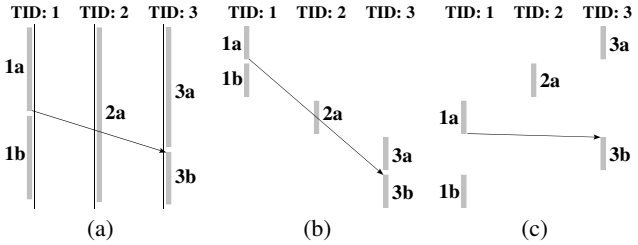


Figure 8: Missing a legal segment reordering. The segments of a happens-before graph (a) are executed using the SID (b) and BID (c) algorithms.

4.3.3 Light64 Active Versions

There are some states in the State Tree that the operation of a conventional systematic tester does not re-execute. Specifically, if it uses DFS exploration, such states are those with zero or one child; if it uses BFS, it can be shown that they are those with no children. Unfortunately, some of the segments in the program may only be executed when such states are reached. If Light64 Active simply reuses the re-executions of a conventional systematic tester, it will not re-execute some of the segments and, therefore, miss any races in such segments.

As an example, consider Figure 2(a), which uses DFS. A conventional systematic tester only re-executes states S2, S3, and S10. Then, Light64 Active would not be able to flip the races in the segments beyond such states, namely B2 and A3.

To detect races in these segments, we can increase the number of re-executions at the cost of a higher overhead. Consequently, we propose three versions of Light64 Active, each with a different tradeoff between probability of race detection and execution overhead. We describe them in the order of increasing race detection probability and execution overhead.

ActiveNO never forces any additional re-execution. This is the version that we have implicitly described so far.

ActiveFIN forces the re-execution of only the final states, namely those corresponding to the termination of the program execution. They are states S5 and S7 in Figure 2(a). Since they have no children, they would never be re-executed otherwise. Re-executing them provides the opportunity to reorder many segments.

ActiveFULL forces the re-execution of every explored state that will not be automatically re-executed by the conventional systematic tester. This scheme maximizes the race detection probability but also has the highest overhead.

Note that the number of additional re-executions needed may also be influenced by the number of threads in the program or the exploration strategy. Specifically, executions with more threads imply more children per node and, therefore, fewer needed additional re-executions under DFS. Moreover, if the exploration strategy follows random exploration rather than the SID algorithm, Light64 Active may choose to perform two re-executions for each state: one following the SID algorithm and one the BID algorithm.

Configuration	Description
Plain	Conventional systematic tester following the design principles of CHESS [15]. Has no race detection capability
Plain+RD	Conventional systematic tester following the design principles of CHESS [15]. Runs a classical precise happens-before race detector
ActiveNO	Plain plus Light64 Active with no forcing of re-executions (Section 4.3.3)
ActiveFIN	Plain plus Light64 Active with forcing re-executions of the final states (Section 4.3.3)
ActiveFULL	Plain plus Light64 Active with forcing re-executions of all the states that are not already re-executed by the systematic tester (Section 4.3.3)
Passive	Plain plus Light64 Passive (Section 4.2)

Table 1: Evaluated configurations.

Finally, there may be some merit in a hybrid solution with mostly Light64 Passive operation and selective application of Active in some key paths.

4.3.4 Reducing the Number of Off-Line Analysis

When the path to a state is found to be racy, Light64 stores away the log of the path to be analyzed offline. For example, assume that Light64 is re-executing state S2 in Figure 2(a) and finds a race between segments A1 and B1. Then, Light64 saves the log for S2 for off-line analysis.

Unfortunately, due to the exploration process, it is likely that Light64 will save many logs with the same dynamic race. For example, assume that we use Light64 ActiveFULL in BFS and that there is no other race in the program. When Light64 re-executes each of states in the S2 subtree, it will detect a race in each and every case and, not knowing that it is the same dynamic race every time, it will save the log for each of these states for off-line analysis. This is wasteful.

Note that this problem also occurs in DFS explorations. Specifically, for the same example, Light64 ActiveFULL detects a race in the path to S5 and saves a log. As Light64 proceeds to explore S4 next, it will not save a log for S4 to minimize redundant analysis, but it will save the log for S7 with the same race.

To avoid this problem and reduce the number of racy logs analyzed offline by orders of magnitude, we propose the optimization that we call *Serial*. Specifically, when a path with a racy segment interleaving is found, after saving the log, Light64 marks the interleaving as *serial*. This means that all the paths being explored in the future that have the first path as a subpath, will not reorder the segments in the subpath. This prevents these future paths from detecting the same race again. To see why, consider the first example given. After re-executing S2, Light64 marks A1–B1 as serial. When Light64 later executes, say, the path to S7, Light64 will keep the order A1–B1 in both the initial and replayed executions, and only re-order segments A2, B2, and A3. Therefore, the re-execution of S7 will not find again the same race and will not save another log for offline analysis.

Note that this optimization will miss the race instances that have one of the racing accesses inside the serial subpath and the other access outside of that subpath. For instance, in the example given, the re-execution of S7 will miss any race between A1 and B2.

5. EVALUATION

5.1 Experimental Setup

We model our lightweight hardware system using PIN [10] and implement a systematic tester following the design principles of CHESS, a state-of-the-art tool used in industry [15]. The segments are the dynamic code sections between synchronization operations or volatile flag access. The exploration follows classical DFS as in other studies [3, 4, 30, 34].

We evaluate the performance and race detection capabilities of the six configurations shown in Table 1: a conventional systematic tester with no race detection capability (*Plain*), Plain with a classical precise happens-before race detector that can find all races but with an extremely-high runtime overhead (*Plain+RD*), Plain augmented with one of the three Light64 Active versions (*ActiveNO*, *ActiveFIN*, or *ActiveFULL*), and Plain augmented with Light64 Passive (*Passive*). We evaluate the performance overhead of Light64 by comparing with Plain and evaluate the detection capability by comparing with Plain+RD. We use instruction count as performance metric and consider that our minor hardware addition has no significant impact on hardware performance. In addition, we estimate the overhead of a software-only implementation of Light64, which would be used in the absence of hardware support.

Our evaluation uses all the applications in SPLASH-2. As customary in systematic testing [14,15,23,29,34], the experiments use small inputs and a small number of threads, which is a pragmatic way to combat state space explosion. For each application, we test four code versions. One is the original SPLASH-2 code. Each of the other three is modified by randomly removing one or more static synchronization operations from the original code, similarly as in other studies on hardware-based race detection [16,21,22,35]. We run each of the four versions with two and four threads, for a total of 576 experiments. To cover the case of bounded search [4, 14, 15], we limit each exploration to a total of up to 50,000 states.

5.2 State Space Characterization

Table 2 gives the characteristics of the explorations. Columns 2-7 correspond to the original SPLASH-2 code, while columns 8-13 show the mean of the three versions modified by inserting races. For each application, the *Distinct states* column gives the number of different states explored during systematic testing and is a measure of testing coverage. We see that our experiments cover a substantial number of states and a wide range of state spaces, ranging from small (FFT, LU) to large (Water-NS, Volrend, Raytrace). Small state spaces are due to extensive use of barriers, which facilitate the pruning of the search, while locks usually lead to large state spaces. The state spaces for the original codes are typically larger than those for the modified versions, since removing synchronization operations decreases the number of possible interleavings.

The *Already seen states* column shows the number of times the exploration encountered an already seen state and thus pruned the search (Section 2.1). This is a measure of how effective the pruning technique is, which is a key factor in performing efficient systematic testing. *DFS stack depth* is the longest path in the State Tree, namely the maximum number of synchronization operations performed during an execution path. The depths of the DFS stacks are large and could lead to huge state spaces (e.g., 2 threads with a stack of 27 as FFT could have 2^{27} different interleavings). However, not all interleavings are possible due to synchronizations (e.g., barriers for FFT, or when a thread is blocked on a lock).

Appl.	Original application						Mean of 3 versions with inserted races					
	2-thread runs			4-thread runs			2-thread runs			4-thread runs		
	Distinct states	Already seen states	DFS stack depth	Distinct states	Already seen states	DFS stack depth	Distinct states	Already seen states	DFS stack depth	Distinct states	Already seen states	DFS stack depth
Barnes	6141	2684	44	20907	29094	62	3854	2080	39	21709	28292	54
Cholesky	4047	3615	113	18201	31800	179	1849	1593	78	18008	31993	144
FFT	39	13	27	195	221	53	21	7	15	105	119	29
FMM	1809	1429	68	18193	31808	108	1153	947	55	18076	31925	87
LU	54	18	37	270	306	73	12	4	9	60	68	17
Ocean	1369	539	77	22161	27840	153	910	363	67	21983	28018	135
Radiosity	26643	23358	1927	22649	27352	2069	26635	23366	1777	22542	27459	1898
Radix	6047	1704	77	22559	27442	172	4797	1618	51	22746	27255	134
Raytrace	29612	20389	1387	18765	31236	1399	28672	21329	679	22069	27932	710
Volrend	30732	19269	69	18286	31715	161	30835	19166	64	19384	30617	116
Water-NS	34644	15357	83	21473	28528	149	29911	16019	72	22453	27548	134
Water-SP	15589	6815	55	21447	28554	109	1711	863	41	22202	27799	90
MEAN	13061	7933	330	17092	24658	391	10863	7279	246	17612	24085	296

Table 2: Characteristics of the state space exploration. For each application, columns 2-7 correspond to the original SPLASH-2 code, while columns 8-13 show the mean of the three versions modified by inserting races.

5.3 Race Detection Capability

Table 3 shows the race detection capability of various configurations. The *Races* columns show the number of static races found, where each race is a triple of an address being accessed and two instruction pointers of the instructions issuing the accesses. Note that the same instructions can access a large number of different addresses, e.g., for array accesses, and thus some applications such as FFT and LU have a large number of inserted races. Because our systematic tester switches at volatile flags, like in CHESS [15], accesses to volatile variables do not count as races.

The *Plain+RD* rows show the total number of races in our experiments. Plain+RD is a precise happens-before race detector that finds all races for the explored executions. In practice, software schemes such as Plain+RD can have up to one or even two orders of magnitude overhead in execution time. This makes them impractical to always run in systematic testers (Section 2.1).

Comparing the *Races* entries of the three versions of Active to Plain+RD, we can see that Active is *highly accurate*. It misses races in only a few cases (which are marked with $\star$ in the *Races* columns). It can be shown that, on average, ActiveNO, ActiveFIN and ActiveFULL detect 93%, 96% and 97% of the races respectively. Passive is less accurate than Active, detecting on average 89% of the races. Even so, the number of missed races is very small and can be acceptable (especially when taking into consideration the extremely low overhead of Passive). We compute these numbers by first computing the percentage of detected races per application and then computing their mean.

The few races missed in Table 3 are due to some of the reasons we outlined earlier in the paper. For example, the *Serial* optimization of Section 4.3.4 causes each Active version to miss four races in two Radix experiments. The races missed by both the Active versions and Passive in Volrend are the special case of *benign* races that we described in Section 3.2. Specifically, in an initialization function, all threads write the same value to a location, instead of just one thread doing the initialization. This is benign since the results are the same, and Light64 does not detect the races because the program execution history is unaffected. ActiveNO misses races in one run of modified LU because these races are between segments leading to final states, which are not re-executed for ActiveNO. Finally, as expected, Passive misses races in several applications because some happens-before graphs are not encountered twice by the systematic tester's explorations.

By checking that all races reported by Light64 are also reported by Plain+RD, we confirm that Light64 reports no false positives. As expected, all logs saved offline by Light64 contain races, and all the races in these logs are also detected by Plain+RD.

We also check (not shown in Table 3) whether inserting reads before writes to cover the corner case discussed in Section 3.2 would discover more races. For each Light64 variant, we try an additional set of experiments, which never detect any new races. Therefore we believe that the additional complexity of inserting reads before writes is not justified.

The *Logs analyzed* columns show the number of logs for which a precise data race detection is run. The numbers for Light64 are much smaller than the numbers for Plain+RD. In the cases of original codes with no races, the numbers for Light64 are 0, since it quickly detects that there is no need to run the precise analysis, while Plain+RD always has to run the precise analysis. For runs with races, the ratio of analyzed logs for Plain+RD over Light64 ranges from around 1.5X (for LU and Passive with four threads) up to 9,526X (for original Ocean with four threads).

The *Logs no opt.* columns show the number of logs that would be analyzed by the Active versions without the optimization of Section 4.3.4. The ratio of logs analyzed without and with the optimization is typically very high, reaching nearly 10,000 for the original Ocean code. This shows that the optimization is highly effective.

5.4 Runtime Overhead

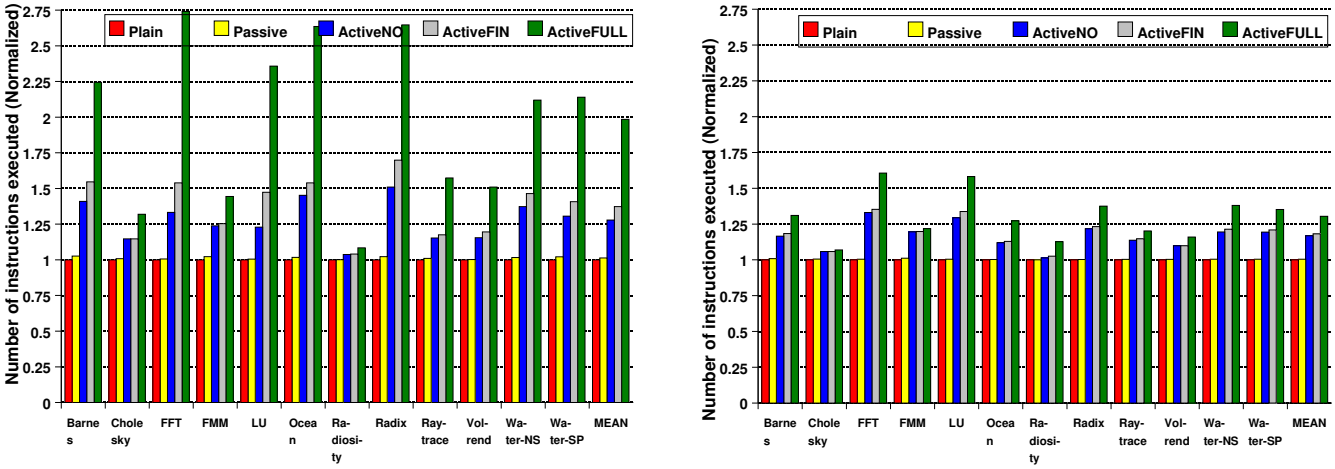
Light64 incurs only a *small runtime overhead*. Figure 9 shows the number of instructions executed by each of the Light64 configurations normalized to the instructions executed by the conventional systematic tester with no race detection (just Plain, not Plain+RD). For each application, the bars are the average of the four code versions, namely the original SPLASH-2 one plus the three modified with races. The offline analysis of the racy logs with a conventional, precise happens-before race detector is characterized by the *Logs analyzed* columns in Table 3 as described in the previous section and is not included in Figure 9.

From the figures, we see that Passive executes only 1% more instructions. For two-thread runs, ActiveNO and ActiveFIN execute, on average, 28% and 37% more instructions, respectively; for four-thread runs, they execute less than 20% more. Passive has a lower overhead than ActiveNO or ActiveFIN because it only observes the executions and compares the history hashes. In contrast, ActiveNO

Appl.	Configuration	Original application						Mean of 3 versions with inserted races					
		2-thread runs			4-thread runs			2-thread runs			4-thread runs		
		Races	Logs analyzed	Logs no opt.	Races	Logs analyzed	Logs no opt.	Races	Logs analyzed	Logs no opt.	Races	Logs analyzed	Logs no opt.
Barnes	Plain+RD	131	1590	n/a	311	8269	n/a	140	927	n/a	192	7865	n/a
	ActiveNO	131	420	420	311	1254	7471	140	202	318	192	1105	7001
	ActiveFIN	131	420	420	311	1254	7471	140	202	318	192	1105	7001
	ActiveFULL	131	420	980	311	1254	8269	140	202	517	192	1105	7865
	Passive	131	420	n/a	311	3251	n/a	140	315	n/a	192	5051	n/a
Cholesky	Plain+RD	4	215	n/a	4	1105	n/a	16	128	n/a	14	1044	n/a
	ActiveNO	4	72	90	4	59	1097	⋆ 5	42	59	14	51	1028
	ActiveFIN	4	72	90	4	59	1097	⋆ 5	42	59	14	51	1028
	ActiveFULL	4	76	98	4	59	1105	⋆ 11	45	65	14	51	1036
	Passive	4	76	n/a	4	2069	n/a	⋆ 12	55	n/a	14	2099	n/a
FFT	Plain+RD	0	14	n/a	0	92	n/a	28459	8	n/a	42688	50	n/a
	ActiveNO	0	0	0	0	0	0	28459	1	1	42688	10	28
	ActiveFIN	0	0	0	0	0	0	28459	1	1	42688	10	28
	ActiveFULL	0	0	0	0	0	0	28459	1	5	42688	11	33
	Passive	0	0	n/a	0	0	n/a	28459	2	n/a	42688	26	n/a
FMM	Plain+RD	0	164	n/a	0	1690	n/a	16	82	n/a	40	1468	n/a
	ActiveNO	0	0	0	0	0	0	16	4	4	40	72	320
	ActiveFIN	0	0	0	0	0	0	16	4	4	40	72	320
	ActiveFULL	0	0	0	0	0	0	16	11	43	40	72	348
	Passive	0	0	n/a	0	0	n/a	16	4	n/a	40	114	n/a
LU	Plain+RD	0	19	n/a	0	127	n/a	15286	5	n/a	15286	29	n/a
	ActiveNO	0	0	0	0	0	0	⋆ 8201	1	1	15286	7	21
	ActiveFIN	0	0	0	0	0	0	15286	1	1	15286	7	21
	ActiveFULL	0	0	0	0	0	0	15286	1	4	15286	8	25
	Passive	0	0	n/a	0	0	n/a	15286	3	n/a	15286	20	n/a
Ocean	Plain+RD	2	416	n/a	2	9526	n/a	24	277	n/a	45	9254	n/a
	ActiveNO	2	4	88	2	1	8369	24	4	62	45	83	8155
	ActiveFIN	2	4	88	2	1	8369	24	4	62	45	83	8155
	ActiveFULL	2	4	396	2	1	9526	24	5	264	45	83	9254
	Passive	2	4	n/a	⋆ 0	0	n/a	24	6	n/a	⋆ 1	868	n/a
Radiosity	Plain+RD	7	860	n/a	12	7779	n/a	12	901	n/a	16	8114	n/a
	ActiveNO	7	487	770	12	1	7167	⋆ 11	510	749	16	38	7447
	ActiveFIN	7	487	770	12	1	7167	⋆ 11	510	763	16	38	7465
	ActiveFULL	7	494	860	12	1	7779	⋆ 11	514	901	16	38	8114
	Passive	7	292	n/a	0	0	n/a	12	425	n/a	⋆ 11	135	n/a
Radix	Plain+RD	0	1450	n/a	0	7039	n/a	7	1166	n/a	16	7491	n/a
	ActiveNO	0	0	0	0	0	0	7	77	427	⋆ 15	171	5700
	ActiveFIN	0	0	0	0	0	0	7	77	427	⋆ 15	171	5700
	ActiveFULL	0	0	0	0	0	0	7	77	1108	⋆ 15	171	6492
	Passive	0	0	n/a	0	0	n/a	7	283	n/a	16	844	n/a
Raytrace	Plain+RD	3	1051	n/a	7	7319	n/a	152	766	n/a	88	3973	n/a
	ActiveNO	⋆ 1	160	160	7	2664	7034	⋆ 149	246	393	⋆ 87	1363	3675
	ActiveFIN	3	321	321	7	2674	7036	⋆ 151	361	512	⋆ 87	1422	3730
	ActiveFULL	3	481	801	7	3032	7236	⋆ 151	369	758	⋆ 87	1663	3972
	Passive	3	480	n/a	7	3047	n/a	152	6962	n/a	88	1792	n/a
Volrend	Plain+RD	45	5576	n/a	44	3911	n/a	46	5786	n/a	43	4821	n/a
	ActiveNO	45	267	1644	44	121	3182	46	312	1872	⋆ 30	81	2123
	ActiveFIN	45	267	1644	44	121	3182	46	312	1872	⋆ 30	81	2123
	ActiveFULL	45	281	2461	44	121	3318	46	328	2728	⋆ 30	81	2213
	Passive	45	364	n/a	44	1211	n/a	46	320	n/a	⋆ 30	807	n/a
Water-NS	Plain+RD	0	9611	n/a	0	8803	n/a	53	6207	n/a	69	8193	n/a
	ActiveNO	0	0	0	0	0	0	53	508	1783	69	886	7173
	ActiveFIN	0	0	0	0	0	0	53	508	1783	69	886	7173
	ActiveFULL	0	0	0	0	0	0	53	632	4688	69	1144	8193
	Passive	0	0	n/a	0	0	n/a	41	2424	n/a	⋆ 52	3815	n/a
Water-SP	Plain+RD	0	4388	n/a	0	8806	n/a	184	422	n/a	162	8507	n/a
	ActiveNO	0	0	0	0	0	0	184	79	190	162	416	7473
	ActiveFIN	0	0	0	0	0	0	184	79	190	162	416	7473
	ActiveFULL	0	0	0	0	0	0	184	82	418	162	416	8507
	Passive	0	0	n/a	0	0	n/a	184	125	n/a	⋆ 116	2765	n/a

Table 3: Characterization of race detection. For each configuration, columns 3-8 correspond to the original SPLASH-2 code, while columns 9-14 show the mean of the three versions modified by inserting races.

and ActiveFIN have to manage the segment execution ordering in the path re-executions. Moreover, ActiveFIN forces additional re-executions. ActiveFULL has the highest overhead because it forces the re-execution of all the states with less than two children.



(a) 2-thread runs

(b) 4-thread runs

Figure 9: Number of instructions executed by each of the Light64 configurations normalized to the instructions executed by the conventional systematic tester with no race detection.

Comparing the overhead results from Figure 9 with the accuracy results from Table 3, we consider ActiveFIN to offer the best trade-off: it is faster than ActiveFULL (but finds just 1% fewer races), and it finds more races than ActiveNO (but is just slightly slower). ActiveFIN detects 96% of races and it incurs only 20%-40% overhead. On the other hand, Passive is the best design point for tasks that require *extremely low overhead* race detection but allow some missed races.

To roughly estimate the overhead of a software-only implementation of Light64, we compare the total running times that our systematic tester implementation takes for various configurations. The base configuration is the regular systematic tester with no race detection and no PIN instrumentation. The other configurations use PIN to dynamically add instructions that compute hashes, mimicking the behavior of the Light64 History register. Note that these configurations include all the fixed overheads of PIN as well.

Table 4 shows the overheads over the base configuration. Note that the data does not include the 2-thread experiments or any of the experiments with FFT and LU, which have small state spaces. The reason is that the data is dominated by the PIN overheads. We see that the mean overhead ranges from 7x to 9x, which is acceptable for a software race detector with high detection capability and no false positives. Like in the hardware evaluation of Figure 9(b), the overhead increases from Passive to ActiveNO to ActiveFIN and to ActiveFULL.

Application	Passive	Active NO	Active FIN	Active FULL
Barnes	6.7x	7.5x	7.7x	8.4x
Cholesky	8.1x	8.3x	8.5x	8.7x
FMM	6.1x	6.9x	7.1x	7.0x
Ocean	4.7x	5.2x	5.2x	5.8x
Radiosity	7.2x	7.1x	7.3x	7.9x
Radix	5.1x	6.0x	6.2x	6.8x
Raytrace	3.0x	3.5x	3.6x	3.5x
Volrend	8.5x	9.5x	9.6x	11.2x
Water-NS	8.6x	10.1x	10.6x	11.6x
Water-SP	13.7x	15.1x	14.3x	16.1x
MEAN	7.2x	7.9x	8.0x	8.7x

Table 4: Estimated overheads of the software-only version of Light64 for runs with four threads.

6. RELATED WORK

There is a growing body of research on systematic testing of parallel (and distributed) code [3, 4, 6, 7, 14, 15, 23, 29–34]. Systematic testing can offer high coverage guarantees and holds promise to detect many types of bugs, as shown for several complex (and safety critical) code bases such as operating systems [15, 30], access servers [7], and distributed execution engines [15]. Light64 contributes by enabling systematic testers to run low overhead data race detection.

Several race detection approaches have been proposed, either in software (e.g., [17, 18, 24–26]) or in hardware [13, 16, 21, 22, 35]. The hardware approaches provide small runtime overhead, which makes them suitable for production runs. However, they can require significant hardware. For example, CORD [21] and HARD [35] increase the cache size, while SigRace [16] adds a race detection module, and ReEnact [22] uses TLS support. In contrast, Light64 requires only a 64-bit History register per core.

Light64 also differs from the other hardware schemes in terms of false negatives and positives. The other schemes have limited storage capabilities in the cache, tags, or race detection module, which limits their ability to detect races in long execution windows. Additionally, HARD has false positives since it uses lockset-based race detection [35]. On the other hand, Light64 is not limited to a certain window because its hashing can track arbitrarily long executions.

Several recent software techniques [2, 11, 19, 26] propose faster detection of concurrency bugs at the cost of missing some bugs. Some techniques [2, 19, 26] use random delays to perturb interleavings and do not offer testing coverage guarantees. RaceFuzzer [26] focuses on one potential data race at a time, confirming whether it is indeed a real race. Unlike Light64, it does not monitor all accesses for races or attempt to find new races. LiteRace [11] uses sampling to monitor only some memory accesses; it reduces the overhead of software-only race detection but may miss many races.

7. CONCLUSIONS

This paper has presented Light64, a novel technique to detect data races during systematic testing that has both small runtime overhead and very lightweight hardware requirements. The key

observation is that, under systematic testing, two different thread interleavings that have the same happens-before graph but a small deviation in the execution history of some thread very likely have a flipped data race. Light64 collects the hashed execution histories and, if they differ, saves an execution log which is later used to precisely detect where the data race is. To efficiently summarize histories, Light64 requires only a 64-bit register per core. Such a design trivially supports virtualization and process migration.

We have evaluated several Light64 configurations using systematic testing experiments on the SPLASH-2 applications with and without inserted races. On average, the recommended Light64 ActiveFIN mode detected 96% of the data races with an increase in instruction execution of only 20%-40%. The low-overhead Light64 Passive mode incurs only a 1% slowdown while detecting on average 89% of the data races. Overall, Light64 keeps the instruction execution overhead much lower than existing software-only techniques, and requires significantly less hardware extensions than existing hardware techniques. These positive results show that Light64 effectively enables systematic testers to run data race detection at all times.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by the National Science Foundation under grants CCF 07-46856, CNS 07-20593, and CCR 03-25603; Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC); Sun Microsystems under the University of Illinois OpenSPARC Center of Excellence; and a gift from IBM.

9. REFERENCES

[1] T. Ball. Personal communication, May 2009.

[2] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPOPP*, June 2005.

[3] D. Bruening and J. Chapin. Systematic testing of multithreaded programs. Technical report, MIT/LCS Technical Memo, LCSTM 607, May 2000.

[4] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, January 1997.

[5] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White. Formal analysis of the remote agent before and after flight. *5th NASA Langley Formal Methods Workshop*, June 2000.

[6] G. J. Holzmann, R. Joshi, and A. Groce. Tackling large verification problems with the Swarm tool. In *SPIN*, August 2008.

[7] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2), 2000.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[9] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2), February 1980.

[10] 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*, June 2005.

[11] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI*, June 2009.

[12] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, August 1991.

[13] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS*, April 1991.

[14] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, December 2002.

[15] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, December 2008.

[16] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *ISCA*, June 2009.

[17] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, June 2007.

[18] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPOPP*, June 2003.

[19] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[20] K. Poulsen. Software bug contributed to blackout, February 2004. http://www.securityfocus.com/news/8016.

[21] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *HPCA*, February 2006.

[22] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, June 2003.

[23] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV*, July 2005.

[24] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the Intel Thread Checker race detector. In *ASID*, October 2006.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, October 1997.

[26] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, June 2008.

[27] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9), September 1990.

[28] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2), April 1988.

[29] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *SPIN*, August 2000.

[30] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2), April 2003.

[31] D. S. Vjukov. Relacy race detector, 2009. http://groups.google.com/group/relacy.

[32] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, April 2009.

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

[34] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Runtime model checking of multithreaded C/C++ programs. Technical report, UUCS-07-008, 2007.

[35] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, February 2007.