LightPlay: Efficient Replay with GPUs

Min Feng[1], Farzad Khorasani[2], Rajiv Gupta[2], and Laxmi N. Bhuyan[2]

[1] NEC Laboratories America
[2] University of California, Riverside**

Abstract. Previous deterministic replay systems reduce the runtime overhead by either relying on hardware support or by relaxing the determinism requirements for replay. We propose LightPlay that fulfills stricter determinism requirements with low overhead without requiring hardware or OS support. LightPlay guarantees that the memory state after each instruction instance in a replay run is the same as in original run. It reduces logging overhead using a lightweight thread local technique that avoids synchronization between threads during the recording run. GPUs are used to efficiently identify the memory ordering constraints that produce the same memory states before the replay run. LightPlay incurs low space overhead for logging as it only stores the part of log where data races occur. During the logging run LightPlay is 20x–100x faster than logging the total order and requires only 1% space overhead.

1 Introduction

The ability to replay a program's execution plays an important role in developing software. Software often fails due to bugs that are difficult to locate. To find the root cause of the bug, developers need to reproduce the bug and observe its manifestation. Replay systems record and reproduce program execution and this capability has proven to be useful in debugging [8, 17, 18] and fault tolerance [4].

Many traditional replay systems [9, 5] record a multi-threaded program's input and the order of shared-memory accesses to achieve deterministic replay on a multiprocessor system. However, logging the shared-memory accesses incurs huge overhead since the execution of these memory accesses needs to be serialized. Many techniques have been proposed to reduce the overhead of logging and replaying shared-memory accesses on multiprocessors. Most of them require hardware modifications to record cache coherence events [15, 6, 13, 20, 22, 14]. LEAP [7] is a software replay system that uses JAVA source code information to reduce the locking overhead used for serializing memory accesses. To make replay more accessible, other efforts have been directed towards reducing the logging overhead using purely software techniques. These techniques relax the determinism requirements so that less information needs to be recorded during the logging phase. Many of the software replay systems [19, 12, 16, 1, 24, 21] guarantee *external determinism* or *output determinism*. *External determinism* only ensures reproducing identical program state at certain execution points in the replay run while *output determinism* just promises that the same values are sent to output devices such as screens, networks, and disks. There are also software

** This work is supported by NSF grants CNS-1157377 and CCF-0905509 to UCR.

replay systems [11, 10] that target even more relaxed determinism requirements, e.g., *failure determinism*. With failure determinism, they only guarantee that the same error states are produced in replay runs. *For these software replay systems, while the output replay may reproduce the same results, the root cause of the error may not be preserved or even reproduced. Therefore, if we use a replay run for debugging, we may not be able to find the original root cause of the bug.*

	Thread 1		Thread 2
1	A = 2;	1	B = 2;
2	...	2	C = add(A,B);
3	A = 3;	3	print(C);

Fig. 1. Example: failures may not be reproduced under relaxed determinism.

Figure 1 shows a code example where the failure may not be reproduced. In the example, thread 2 collects values from all threads, calculates the summation, and prints the result. Consider a program run that has the following execution order: T1:1, T2:1, T2:2, T1:3, and T2:3, but the output is 5 due to a bug inside function add(). To replay this execution, an output deterministic replay system may produce an execution in which the output is still 5, but the execution order is T1:1, T1:3, T2:1, T2:2, and T2:3. 3 plus 2, however, is 5 and thus the replay run does not show any fault. Developers cannot find the bug using this replay run.

In this paper, we propose LightPlay, which is a deterministic replay system designed to log and replay multithreaded programs that are executed in parallel on multiprocessors. The system does not require any modification to the hardware or the OS. LightPlay delivers a stricter determinism requirement - *internal determinism*. We guarantee that the internal memory state after each instruction instance in a replay run is the same as in the original run. By keeping the internal states the same, we can ensure that the root cause of the bug is unchanged in the replay run. However, achieving *internal determinism* using prior software techniques is costly. Recording either total orders [9] or load values [2] requires serializing shared memory accesses, atomic execution of instrumented code, and/or data privatization. We have developed a set of techniques to make LightPlay efficient. In LightPlay, the logging overhead is reduced via use of lightweight thread local technique which is designed to avoid additional synchronization between threads in the recording runs. Since the thread interactions are not recorded, it is necessary to search for an equivalent multi-threaded execution before the replay run. To enable searching, execution is divided into timeslices and values read and written by shared-memory accesses during each interval are roughly recorded at runtime without serializing the accesses. In this way, we reduce the logging time by migrate the runtime overhead from the recording run to the replay run. To further improve the search performance, we use GPUs to perform the search in parallel. The GPU uses recorded values for a slice to search for an ordering of shared-memory accesses that produces the same memory state. The ordering recovered by the GPU is then saved and used during deterministic replay. Our experiments show that LightPlay is 20x–100x faster than logging the total order in the logging run and requires only 1% space overhead. The search procedure on the GPU is also shown to be efficient, causing less than 30x slowdown for most PARSEC benchmarks.

2 The LightPlay System

LightPlay is a deterministic replay system that records and reproduces execution of multithreaded applications. It uses a purely software logging mechanism which, unlike other software solutions that introduce high runtime overhead, is very light-weight. The recorded trace is composed of a set of incremental checkpoints. Therefore, the replay can be begun from any point in the execution. However, the trace captured via logging cannot be directly used to reproduce the execution as thread interactions are not recorded in order to achieve light-weight logging. This issue is addressed by searching for an execution that has no observable difference from the original execution. To perform the search efficiently, we propose to exploit the massive parallelism offered by GPUs. After finding the desired execution, it is replayed using a uniprocessor.

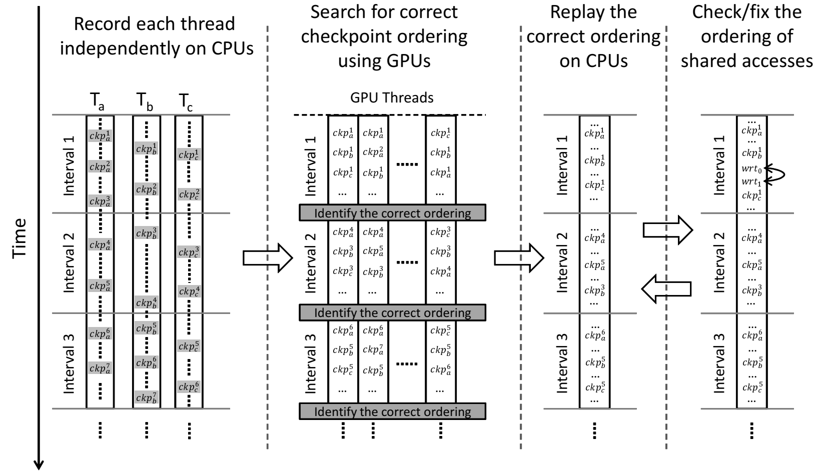

Fig. 2. System overview.

Figure 2 shows the four steps of LightPlay. First, the logging of each thread in the multithreaded application is performed independently via incremental checkpoints. The program's execution is divided into intervals referred to as time slices such that they contain *nearly equal* numbers of checkpoints. In the second step, the system uses GPUs to search for a correct checkpoint ordering before replaying the execution. The search can be done offline or it can be done online (i.e, as the program executes on the CPUs, the GPU can search for the ordering). In step three, the system infers the thread schedule on a uniprocessor that reproduces the execution. Checkpoint ordering sometimes does not lead to a correct replay. If this happens, in step four, the system uses the error information to adjust the thread schedule. The third and fourth steps are repeated until a correct replay is achieved.

2.1 Logging

Our logging mechanism is designed to be both lightweight (i.e., it introduces little execution time overhead) and entirely implemented in software. Traditionally, to record a multithreaded execution without hardware support, synchronization

points are injected for collecting: (1) ordering of shared memory accesses (i.e., thread schedule); and/or (2) input and output values of shared memory accesses. This additional synchronization introduces huge runtime overhead. Our logging mechanism collects neither the ordering nor the accurate input/output values of shared memory accesses. Instead it records each thread independently and only inserts lightweight incremental checkpoint sites to record the program state before and after (although not necessarily immediately before/after) shared memory accesses. We then search for an equivalent execution that produces exactly the same values at the checkpoints when performing replay.

Thread 1	**Thread 2**
A = B = 0;	
. . .	
record B;	record A;
A = B + 1;	C = A + 1;
record A;	record C;
. . .	
record A, C;	

Fig. 3. Example showing Checkpoints.

```
ss = ++slice_size; // get the current slice size
if ( ss == T+1 ) { // first thread reaching a new slice
   wait_current_slice_finish();
   checkpointing();
   slice_size -= T;
   notify_other_threads();
}
if ( ss > T+1 ) // other threads wait if the slice is full
   wait();
// shared memory access here;
```

Fig. 4. Timeslicing code.

Inserting optimistic checkpoints. LightPlay introduces lightweight checkpoints using PIN for quick prototyping (static insertion can also be used). The checkpoints are inserted at: (1) before/after each shared memory accesses; and (2) after each time slice. Checkpoints are added before (after) each shared memory access to record the state of the read (written) memory location. We treat all memory accesses other than accesses to the stacks or thread local storage as potentially shared memory accesses. In the example given in Figure 3, we record `B` & `A` before the two memory accesses and `A` & `C` after the accesses. The recorded values may not equal the input and output values of the shared memory accesses since we do not perform the original memory access and their recording *atomically*. This *optimistic* approach is used for reducing overhead. In vast majority of the cases the values will be correctly recorded and in the few cases where they are not, step four of our approach will account for them. Checkpoints are also inserted to record the program state at the end of each time slice to enable ordering of the writes before the end of each time slice. We only record the state of memory locations that have been updated. For example, in Figure 3, `A` & `C` are recorded at the end of the time slice since they have been updated.

Timeslicing. All threads need to be synchronized at the end of each time slice. This synchronization can be done with the use of barriers, as shown in Figure 5(a). However, barriers may greatly slowdown the execution due to resulting stalls. In addition, using hand coded barriers may cause imbalance in time slices, that is, different time slices may have different number of shared memory accesses, making checkpoint ordering costly for large time slices. Our timeslicing scheme avoids using barriers and creates timeslices of equal size. At the end of each timeslice, we use one thread to finish the timeslice and allow the

other threads to continue until encountering new shared memory accesses (see Figure 5(b)). Figure 4 shows the pseudo-code of our timeslicing implementation. Code is inserted before each shared memory access to divide the execution into time slices that contain equal number (i.e., `T`) of shared memory accesses. The increment (i.e., ++) and compound assignment (i.e., -=) operators in the example are made to be atomic using the *atomic fetch-and-add instructions* provided in the x86 instruction set. This ensures if a thread grows the slice size beyond `T`, it waits until the current slice is completed. This improves performance over using barriers since a thread can continue to perform local memory accesses even when another thread is finishing the current time slice.

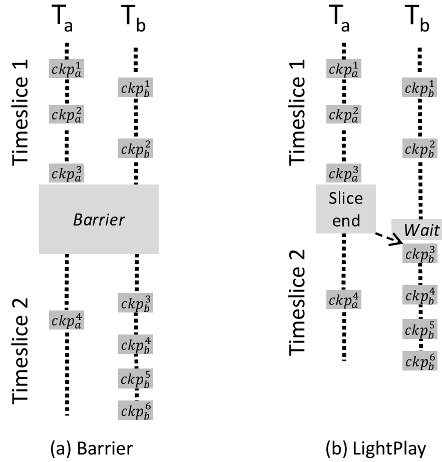

Fig. 5. Barrier vs. our slicing implementation.

Reducing overhead of logging values. Value logging can incur huge space overhead. For example, logging the `streamcluster` benchmark [3] will produce over 10GB of trace per minute. If the trace is constantly dumped onto a disk, logging performance suffers. Therefore, instead of keeping value logs for all slices, we only keep logs for the time slices that have data races. If there is no data race in the slice, the slice can be replayed with any thread schedule. We only need to store two numbers in the log file, each number indicating the number of shared accesses performed by a thread. This optimization does not cause any information loss since all memory accesses in the rest of the time slices can be viewed as local memory accesses which do not require logging. This optimization greatly reduces the trace size since most time slices usually do not have any data race. To further improve performance, we create separate threads to check the existence of data race in parallel with the program execution. Thus, the overhead of data race checking is removed from the critical path of the program execution,

2.2 Checkpoint Ordering

Finding the checkpoint ordering not only reproduces the execution, but it can also help users debug their programs because it helps in understanding of cross-thread interactions, i.e., shared-memory dependencies. We propose to use GPUs to accelerate the search for the correct ordering. The search is performed one

timeslice at a time. During the search, we optimistically assume that each shared memory access and the checkpoints around it were executed atomically. In other words, we treat the checkpoints before and after each shared memory access as one memory instruction, which reads the memory locations recorded before the access and writes the locations recorded after the access. The GPU searches for an ordering of such memory instructions that correctly produces the program state at the end of the slice, which is recorded by the checkpoints. The resultant ordering should satisfy the ordering in each thread in the original execution. The detail of using GPUs for searching will be elaborated in Section 3. The search for the ordering can be done online and offline.

During the above search we optimistically assume that shared memory access and the checkpoints around it were executed atomically. However, this is not the case because we do not enforce atomicity during the logging phase. As a result, we might not be able to find a correct ordering in some cases or the ordering found may not lead to a correct replay. However, this situation is rare and when it does arise, it is handled by fixing the replay as described later in Section 2.4.

2.3 Replay

```
foreach checkpoint_pair (x, y) in sorted_checkpoint_list {
  i = the memory instruction associated (x, y);
  perform i;
  if the output values match y {
    // succeed
    update the memory;
  } else {
    // replay fails
    return error;
  }
}
```

Fig. 6. Pseudo-code for replaying.

Once we obtain the checkpoint ordering, we can reproduce the execution of the time slice on a uniprocessor (see Figure 6). Recall that we treat the checkpoints before and after each shared memory access as one memory instruction in the searching. Therefore, each element of `sorted_checkpoint_list` is a checkpoint pair. During replay, we enumerate the checkpoint pairs according to the ordering and execute the memory instructions associated with the checkpoint pairs. If the output values match the recorded program state after the instruction, we have successfully replayed the instruction and go to the next pair of checkpoints. Otherwise, the replay has failed and our system will try to fix the replay as described in the next section. If all instructions are replayed successfully, then we have correctly reproduced program behavior via an equivalent execution. The original and equivalent executions produce the same program state at the end of each time slice. In each thread, the two executions reach the checkpoints in the same order and produce the same values. Compared to previous software-based replay systems such as DoublePlay [19], the determinism we achieve is more strict since they achieve external determinism (i.e., the orders of system calls and program states at the end of each timeslice are identical).

```
function fix_ordering(checkpoint_pair(x, y)){
  i = the associated memory instruction;
  foreach checkpoint_pair (s, t) after (x, y){
    j = instrn associated with (s, t);
    if ( t updates the location in x and
      j can be moved before i ) {
      perform i using the values in t;
      if output values matches {
        move (s, j, t) before i;
        return Succ;
      }
    }
  }
  foreach checkpoint_pair (s, t) {
    j = instruction associated with (s, t);
    if ( t matches y and
      j can be moved after i ) {
      move (s, j, t) after i;
      return;
    }
  }
  return Fail; }
```

Fig. 7. Fixing the instruction ordering in the presence of mismatched output values.

```
foreach ordering in ordering_list {
  i = first memory instrn in instr_list;
  while (i in instr_list) {
    do {
      if i not in ordering {
        find checkpoint_pair (x, y)
          associated with i;
        insert i after x in ordering;
      } else {
        move i to the next position
          in ordering;
      }
    } while (i break the def-use chain
      in ordering and
      i is before y);
    if (i inserted successfully)
      i = next instrn in instr_list;
    else
      i = previous instrn in instr_list;
  }
  if (all instrns inserted successfully)
    return ordering;
}
```

Fig. 8. Searching instruction ordering when no checkpoint ordering is found.

2.4 Fixing the Ordering

Since we do not enforce the atomic execution of each shared memory access and its checkpoints, we may not be able to correctly reproduce the execution in the previous steps. There are two types of errors that might occur: (1) a checkpoint ordering can be found but the output values of a replayed instruction do not match the values stored in the subsequent checkpoint; and (2) no checkpoint ordering can be found. Next we describe how to fix the replay in each case.

Figure 7 shows the algorithm for fixing the memory access ordering when the first case described above arises. There could be two reason behind this: (1) another thread updates the input memory locations immediately before the misreplayed instruction; and/or (2) another thread overwrites the output memory locations immediately after the misplayed instruction. In the first case, we examine all subsequent memory instructions to see if they write to the input memory location. If another instruction does and can be moved before the misreplayed instruction without breaking the existing def-use chain, we try moving it before the misreplayed instruction. If the new output values matches, we then get the correct ordering. Similarly for the second case, we can try moving another instruction immediately after the misreplayed instruction. If the ordering cannot be fixed, the method presented next handles the situation in which no checkpoint ordering can be found.

When no checkpoint ordering can be found, we relax the condition for ordering searching by treating each checkpoint as an independent memory instruction.

If the checkpoint is set before a memory access, it is considered a read. Otherwise it is considered a write. We then send the new set of memory instructions to the GPU for searching ordering. Because in the new search we give up the assumption that a memory access and its associated checkpoints are executed atomically, the position of each memory access in the resultant ordering may not be inferred directly. Figure 8 shows the algorithm to search the ordering of all shared memory accesses given the ordering of checkpoints. The algorithm tries to insert each memory access into positions between the two associated checkpoints until a correct ordering of the shared memory accesses is found.

2.5 Minimal Constraints Construction

Our search algorithm on the GPU is designed to find all correct orders of shared memory accesses. With the information, we are able to identify a minimal set of memory ordering constraints, which can help programmers understand the bug and enable fast parallel replay. We use a simple efficient method to extract minimal constraints between memory instructions by knowing all correct possible orderings for them in a timeslice. Instruction ordering within each thread applies the first set of constraints to the memory instructions. We also consider one of all correct permutations as the reference correct ordering. The sufficient condition for two memory instructions from different threads to be dependent upon each other is that they access the same memory region and at least one of them writes. For pairs of memory instructions that satisfy this condition, if in all the correct orderings the same pattern is observed, we introduce this pattern as the constraint. Otherwise, we introduce the constraint between two instructions using the reference correct ordering.

3 Searching for Ordering via GPUs

We use GPUs for searching the correct checkpoint ordering since the massive parallelism of GPUs is a good fit for searching. The step in Section 2.2 requires that we find the ordering of checkpoint pairs. We begin by treating the checkpoints before and after each shared memory access as one memory instruction. We need to find an ordering of such memory instructions that correctly produces the program state at the end of the slice. Besides, the resultant ordering should satisfy the ordering in each thread during the original execution. The step in Section 2.4 requires that we find the ordering of individual checkpoints but all the rest remains the same. Therefore the problem we need to solve on the GPU is, how to identify the correct ordering of a set of memory instructions given: (1) the input/output values; (2) the final program state; and (3) the instruction ordering within each thread. To perform searching, we first transfer the data regarding the time slice to the device memory and then each GPU thread works on a different ordering to see if that ordering works for the slice. Because of the massive parallelism available on the GPU, this step is done more efficiently on the GPU. Finally the correct ordering found is transferred to the host.

3.1 Ordering Search

The following three-step procedure is performed on each GPU thread to find correct ordering of memory instructions. Given a time slice, we first generate all

permutations of the memory instructions on the GPU and each GPU thread is assigned one permutation. Second, the thread emulates the memory instructions using the ordering indicated by the permutation. Finally, we verify the program state with recorded state. The details of this procedure are given next.

Generating the permutation. A unique permutation of the memory instructions in a given slice is created according to the GPU thread ID. When the algorithm creates the unique permutation, it respects the instruction ordering within each thread. In other words, our algorithm guarantees that the instruction orderings in the generated permutations match the recorded local instruction orderings. In this way, we eliminate most incorrect permutations of instructions in the generation step, which increases available GPU resources for each permutation, allowing handling of bigger slices with more instructions.

The slice information is stored in three arrays: *instr_list*, *cpu_tid_list*, and *thread_size*. Array *instr_list* stores the instruction IDs. In a time slice, instructions are sorted by their thread IDs and numbered from 1 (as shown in Figure 9(a)). Array *cpu_tid_list* contains the thread ID of each instruction and *thread_size* keeps the number of instructions in each thread. The number of possible permutations is stored in *perm_count*, which is initially set by the following equation where T is the number of threads and *slice_size* is the number of instructions in the slice.

$$perm_count = \frac{slice_size!}{\prod_{i=0}^{T}(thread_size_i!)}$$

Figure 9(a) shows a recorded slice with two threads. In the example, thread 1 contains three instructions numbered from 1 to 3 and thread 2 contains two instructions numbered 4 and 5. Figure 10 gives all the permutations generated by our algorithm. We can see that all permutations match the recorded local instruction orders. Figure 10 shows the step-by-step procedure of generating a permutation on GPU thread 4 using our algorithm. Each step removes an instruction from thread 1 or 2 and assigns it to *my_perm*. The procedure ends when all instructions are assigned to *my_perm*.

One possible way to create all permutations is to create them on the CPU once and then transfer them to the GPU. In this way, aside from the initial workload for moving the permutation table, the number of memory accesses on the GPU increases enormously, degrading the performance. In our approach, each thread creates its own permutation by using its own registers and shared memory and therefore no time is wasted on memory accesses.

Emulating the instructions. The given memory instructions are emulated in the GPU memory. Figure 11 shows how the instructions are emulated. Execution of each instruction is emulated on an array of registers assigned to the thread. Memory instructions are verified one by one. If the instruction is a write, the emulation array (EmuMeM) will be updated using the recorded value. If the instruction is a read, the emulation array is searched for the corresponding value. For each read, we check if the values gotten from the emulation array matches the recorded values. If not, this permutation is wrong and the thread will be killed. At the end, a final memory state is created based on the ordering corresponding to the permutation.

CPU Thread 1	**CPU Thread 2**
I1→I2→I3	I4→I5

(a) A recorded slice with two CPU threads.

GPU Thread ID	**Permutation**
0	I1→I2→I3→I4→I5
1	I1→I2→I4→I3→I5
2	I1→I2→I4→I5→I3
3	I1→I4→I2→I3→I5
4	I1→I4→I2→I5→I3
5	I1→I4→I5→I2→I3
6	I4→I1→I2→I3→I5
7	I4→I1→I2→I5→I3
8	I4→I1→I5→I2→I3
9	I4→I5→I1→I2→I3

(b) Generated permutation on each GPU thread.

Fig. 9. Example of generated permutations.

Iteration	Thread 0	Thread 1	rank	perm_count	instr_list	cpu_tid_list	my_perm
Init.	I1→I2→I3	I4→I5	4	10	{1,2,3,4,5}	{0,0,0,1,1}	{}
#0	I2→I3	I4→I5	4	6	{2,3,4,5}	{0,0,1,1}	{I1}
#1	I2→I3	I5	1	3	{2,3,5}	{0,0,1}	{I1,I4}
#2	I3	I5	1	2	{3,5}	{0,1}	{I1,I4,I2}
#3	I3	ϕ	0	1	{3}	{0}	{I1,I4,I2,I5}
#4	ϕ	ϕ	0	0	{}	{}	{I1,I4,I2,I5,I3}

Fig. 10. Permutation generation on GPU thread 4.

Verifying the final state. Finally the contents of emulation array are compared with the recorded final state of the time slice. If they do not match, the permutation does not work and the thread is killed. If there is a match, the permutation indicates a correct order of shared memory accesses is transferred back to the CPU.

3.2 Optimizations

Slice refinement. To reduce the search space, we apply two levels of refinements to a slice. In the first level of refinement, we remove the local memory accesses from the slice. The second level of refinement splits a big slice into smaller ones to further reduce the searching overhead.

Data transfer. To improve the data transfer performance, multiple CUDA streams should be used. Instead of one, multiple time slices of instructions are processed in parallel with multiple streams. By enqueueing operations of multiple streams in a breadth-first manner, data transfer and searching (i.e., kernel) are performed concurrently. Overlapping of host-device data transfer and kernel execution accelerates the GPU algorithm by improving the throughput. For fast copying of results back to the CPU, we use the host zero-copy memory to hold them. Since the result is accessed just once during the kernel call, using zero-copy method instead of copying the data from the GPU global memory to the CPU memory greatly improves the performance. Since each GPU thread ID in-

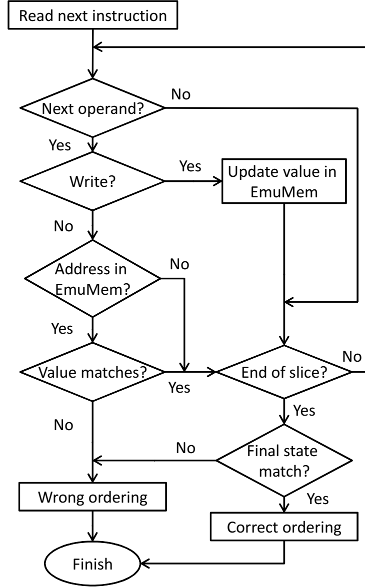

Fig. 11. Flowchart of verifying an ordering.

dicates a specific ordering, we only need to store the thread IDs in the zero-copy memory, which further reduces the data transfer overhead.

4 Evaluation

To evaluate our LightPlay we conducted experiments by running PARSEC benchmark suite [3] on a machine with twelve Intel i7 processors (3.20GHz) and two NVIDIA GeForce GTX 780 cards. Most PARSEC benchmarks (except Blackscholes) have data races between threads. We used the large input set provided with the PARSEC benchmarks for evaluation. All PARSEC benchmarks were run using 4 threads unless specified otherwise. We used two GeForce GTX 780 graphic cards for the experiments with CUDA driver version 5. We used the constant memory on the GPU to store the slices. Accesses to constant memory, unlike regular global memory, are cached. During GPU kernel launches, we used shared memory and registers to improve performance as much as possible. Considering the fact that GeForce GTX 780 has around 48 KBytes of shared memory, we had to choose maximum possible size of a slice so that block size be a multiple of warp size and also each thread block can have a suitable quota of shared memory for operations. We selected 32 as the block size of each GPU kernel call and it confined maximum possible slice size to 18. In the experiments, each GPU uses at least two streams to maximize overlapping between data transfer and kernel execution.

4.1 Logging Performance

Figure 12 shows the logging performance of LightPlay compared to total order recording. Total order recording is implemented by atomically executing an memory instruction with the corresponding recording instructions. We can see that LightPlay is faster than total order recording by at least a factor of 20x.

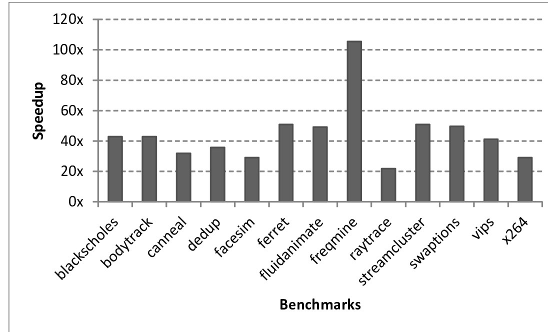

Fig. 12. Logging performance over total order recording.

Benchmark	LightPlay	Total Order Recording
blackscholes	2,624	649,343
bodytrack	10,749	2,577,869
canneal	16,002	3,948,534
dedup	39,729	9,817,656
facesim	315,683	78,124,661
ferret	22,609	5,584,933
fluidanimate	18,311	4,458,619
freqmine	51,016	12,621,886
raytrace	363,978	89,987,985
streamcluster	193,901	5,535,742
swaptions	11,523	2,850,757
vips	34,882	8,632,856
x264	8,155	2,005,157

Table 1. Log size (KB) of LightPlay and Total Order Recording.

Benchmark	Percentage
blackscholes	0.0000%
bodytrack	0.0062%
canneal	0.0019%
dedup	0.0015%
facesim	0.0000%
ferret	0.0011%
fluidanimate	0.0024%
freqmine	0.0002%
raytrace	0.0007%
streamcluster	0.1460%
swaptions	0.0006%
vips	0.0004%
x264	0.0010%

Table 2. Percentage of slices with data races.

Table 1 lists the log size of LightPlay and total order recording. Compared to total order recording, LightPlay does not need to record the trace for all memory instructions. We only record values for slices which contain data races. As shown in Table 2, all benchmarks contain less than 0.002% slices that have data races. Thus, our log size is significantly smaller than that of total order recording. We observed reduction in the log size of over two orders of magnitude.

4.2 Search Performance

Figure 13 shows the search performance for most PARSEC benchmarks (except `blackscholes` and `streamcluster`) using GPUs. In the experiment, we run the benchmarks using 2, 4, and 8 CPU threads. The slowdowns were measured over the original execution time of the same benchmarks. We can see that for most benchmarks, the slowdown caused by the search is below 30x. We do not show `blackscholes` in the figure since it has no slice with data races. The search performance for `streamcluster` is 1000x 2000x because its trace contains 673K slices. The performance for `streamcluster` is still acceptable compared to other search-based replay systems.

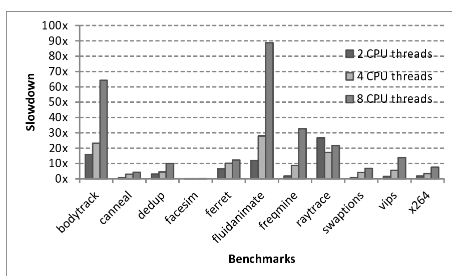

Fig. 13. Search performance using GPU.

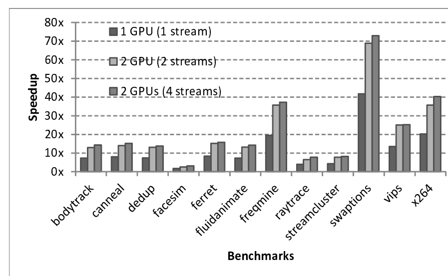

Fig. 14. Search performance compared to a CPU implementation.

In Figure 14, we compare our search performance with a CPU implementation, which runs a similar algorithm. The GPU performance was measured by using 2 GPUs, each with 4 streams. The host-pinned memory was used to help transfer data to the GPUs. We used 8 threads for the CPU implementation. Compared to the CPU implementation, our GPU implementation is on average 28x faster, which demonstrates GPU is a good platform for performing the search. Figure 15 shows a breakdown of the execution time of the GPU kernel. For all benchmarks, over 96% of time is spent on computation. Therefore, our GPU implementaiton has very high computation to data transfer ratio.

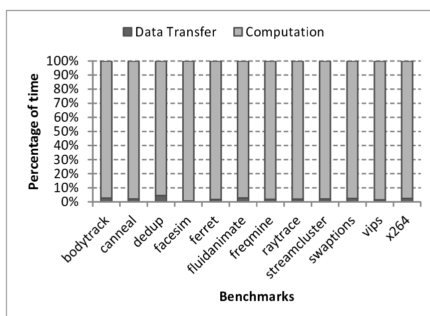

Fig. 15. Breakdown of GPU kernel execution time.

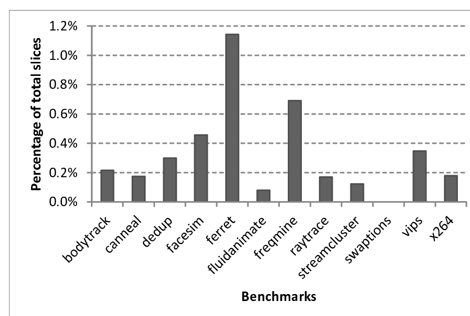

Fig. 16. Percentage of slices that need to be fixed.

4.3 Fixing the Order Performance

Figure 16 shows the percentage of slices that need to be fixed after the search. For all benchmarks, only less than 1.5% of slices require the fixing procedure. The `swaptions` benchmark shows an empty bar since it does not have any slices requiring order fixing.

Figure 17 shows the performance of the fixing procedure. The slowdowns were measured over the original execution time of the same benchmarks. The fixing procedure causes less than 2x slowdown for most benchmarks except `streamcluster`. It takes much less time than the search. For `streamcluster`,

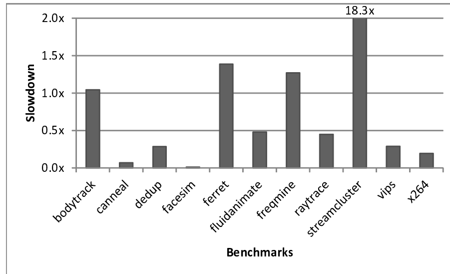

Fig. 17. Performance of order fixing.

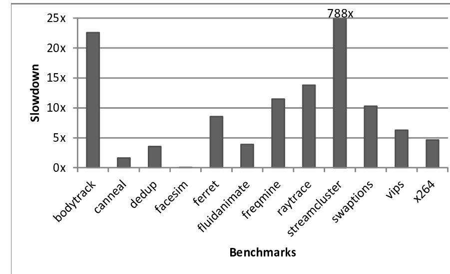

Fig. 18. Performance of constructing the minimal ordering constraints.

the fixing procedure causes around 18x slowdown since it has more than 800 slices that need to be fixed.

4.4 Constructing Minimal Ordering Constraints

Figure 18 shows the performance of constructing the minimal ordering constraints. The minimal ordering constraints were constructed on the CPU. We can see that for most benchmarks, the slowdown caused by the construction is below 25x, which costs less than the search procedure. The constraint construction for `streamcluster` takes longest time, causing 788x slowdown. This is because its trace contains 673K slices. The step of constructing minimal ordering constraints is optional since the constraints are not necessary for replay.

5 Conclusion

We presented a software solution to record and replay multithreaded programs. Our solution does not require recording shared-memory dependences. Each thread only records its own memory instructions locally. In the replay run, GPUs are used to quickly identify the correct total order of shared memory accesses. In the recording phase, memory accesses are organized in time slices to reduce the search space. To reduce the logging overhead, the traces for time slices are selectively saved and the checkpointing at the end of a time slice is done using a parallel thread. The experiments show that logging overhead is very low and GPUs are much faster than CPUs in building correct ordering of shared accesses.

References

1. Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP*, pages 193–206, 2009.
2. Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, pages 154–163, 2006.
3. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, 2008.
4. Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

5. George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.
6. Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA*, pages 265–276, 2008.
7. Jeff Huang, Peng Liu, and Charles Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *FSE*, pages 207–216, 2010.
8. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.
9. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.
10. Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. Offline symbolic analysis to infer total store order. In *HPCA*, IEEE, 2011.
11. Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. Offline symbolic analysis for multi-processor execution replay. In *MICRO*, pages 564–575, 2009.
12. Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS*, pages 77–90, 2010.
13. Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, pages 289–300, 2008.
14. Vijay Nagarajan and Rajiv Gupta. Ecmon: exposing cache events for monitoring. In *ISCA*, pages 349–360, 2009.
15. Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ASPLOS*, pages 229–240, 2006.
16. Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multi-processors. In *SOSP*, pages 177–192, 2009.
17. Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.
18. Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.
19. Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *ASPLOS*, pages 15–26, 2011.
20. Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS*, pages 271–284, 2010.
21. Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, 2010.
22. Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, pages 122–135, 2003.
23. Cristian Zamfir, Gautam Altekar, and George Candea. Debug determinism: The sweet spot for replay-based debugging. In *HotOS*, 2011.
24. Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.