1 Introduction

1.1 Deterministic Record and Replay

Deterministic Record and Replay [5, 6, 4] addresses the nondeterministic nature of memory access interleavings in executions of parallel codes. The loss of determinism introduces complexities to common processes such as program analysis or debug of software. This is because two accesses to one location by two processors when at least one of them is writing is a data race. For example, consider a program which has a bug due to insufficient fencing on a TSO machine. Without a fence, the processor is allowed to relax the $W \rightarrow R$ ordering when advantageous. A common case is that, if the write results in a cache miss, the processor can issue a read before the write completes, i.e. retire from write buffer. This could be a problem, if for example, the write was to acquire a lock. In that situation reads may be issued that return stale data, causing the program to crash. Subsequent executions of the program may or may not exhibit this behavior because their lock write may hit in the cache, the read may not be reordered with the write, or the read may return consistent values when reordered with the write.

Deterministic Replay for TSO machines was developed in a previous work by Gilles et al [5]. If a processor is able to record the global processor memory access ordering then a replay system can instrument the code to enforce those orderings. If the processor can capture the complete set of RAW, WAW, and WAR dependencies then it will capture this ordering. Once the orderings are known, then a Causal Precedence Graph (CPG) can be created. The CPG is a Directed Acyclic Graph where the nodes are chunks of a program execution and an edge (A,B) implies that chunk A must occur before chunk B. For our project we focussed on the work done by Gilles et al, as it reflects the state of the art with regards to RnR on TSO machines.

1.2 QuickRec

One efficient solution to detect these races is by using Bloom Filters, a probabilistic data structure, to track addresses written and read by a processor during execution. A Bloom Filter can be thought of as a very large bit vector, where elements are inserted by setting a number of bit locations based on hashes of the elements. As such, the Bloom Filter guarantee the absence of False Negatives when testing for set inclusion. This enables efficient detection of the above mentioned conflicts by simply testing the appropriate Bloom Filters of other processors before completing a read or a write. As such, Gilles et al. used Bloom Filters as the data structure for tracking memory accesses in their Memory Race Recorder (MRR). They place the MRR logic in between the private L1 and the shared L2. If the L2 is inclusive, there is a guarantee that the L2 will snoop the L1 to invalidate cache lines due to writes from other processors. This snoop can be used to detect the inter-processor RAW, WAW, and WAR conflicts. While this system solves the problem of deterministic replay, it introduces the problem of very slow replay speed when compared with the original execution. The reasons are two fold. First the lack of precision of the recording hardware and the second is due to the overheads of enforcing the exact interleaving recorded by the hardware. We will explain this in more detail in the following sections.

2 Motivation

If the CPG is precise then every node N in the CPG must end with the exact memory access that needs to occur before the next task, and must begin with the exact memory access that needs to occur after the previous task. The design of QuickRec does not capture precise information because it would require capturing the full ordering of memory accesses and searching for precise location of the conflicts when
creating a chunk. A few limitations of QuickRec result in the imprecise nature of QuickRec logs. The first is that MRR requires that the L1 is snooped for all memory access requests to intercept accesses and check for dependencies. If the L2 can silently provide the data, then the MRR cannot detect the races. However, cache line evictions may occur due to conflicts, and in those cases the line may be satisfied by the L2. As such, any time a cache line eviction happens, an incorrect ordering will be established between the evicting processor and all the other processors. This false ordering is a source of imprecision in the CPG, because it will create a set of edges between the current chunk and the current chunk of all the other processors.

To guarantee max precision, the MRR, must provide perfect sensitivity and specificity when detecting memory races. Perfect sensitivity implies that the Bloom Filter will detect any address that has been accessed in the past. This will allow the MRR to detect if any address has been written or read by the processor. Perfect specificity implies that for any arbitrary address A, the MRR will always be able to detect that address A has not been accessed by the processor. With perfect sensitivity and specificity precise identification of all RAW, WAR, and WAW conflicts is possible. By design, the Bloom Filter guarantees no false negatives – so it is perfectly sensitive. However, as a consequence of this design, the probability that the Bloom Filter reports inclusion for arbitrary addresses increases as more elements are inserted into the Bloom Filter. Because of the Bloom Filters natural imprecision and because of L1 cache evictions, a large majority of the edges in the CPG are invalid. False dependencies do not impact correctness of the replay but they significantly reduce the amount of replay parallelism available. With our Max-precision algorithm we tackle this problem.

Another source of imprecision is the granularity at which conflicts are detected and enforced in the time domain and in the instruction domain. Consider a RAW conflict where the read occurred much later than the write. The chunks created will only be created once the read is detected. In this scenario, the chunks will contain many victim accesses that are unnecessarily ordered with respect to the write. Replay is designed to execute the exact interleaving of shared memory accesses by enforcing the conservative invariant that no chunk shall begin executing if there exists any chunk with a Time Stamp Difference (TSD) lower than its own. This restriction forces the second chunk to have to wait until the first chunk is completely executed, even though the replay can still maintain equivalence with the recording if the chunk were to start after the completion of source of the dependence in the prior chunk. Ultimately, this causes the loss of ideal chunk overlap due to the ordering of chunks using TSD recorded the end of the chunk. This also causes some false dependencies to be created during replay time, as all chunks must wait until all previous chunks with lower TSDs executed, even if the other chunks are not ordered with respect to the pending chunk. With our Max-replay algorithm we tackle this problem.

3 Improved Replay

In the next few sections we outline our algorithms to first generate memory access interleavings from a parallel hardware run (Section 4), and then application of two techniques - max-precision (Section 5) and max-replay (Section 6) for improving the replay speed of the recorded application. The former technique enables removal of false dependencies between chunks such as those caused by cache evictions. The latter aims to break chunks into smaller chunks such that unnecessary instructions are not ordered by the replayer. We evaluate our results on the SPLASH benchmark suite, and compare our numbers to numbers from QuickRec implementation [5].

4 AIG Construction

4.1 Overview

Address Interleaving Generator (AIG) is a Pintool that produces all memory access details for one execution of a parallel program. Specifically, for each memory read/write, it records the thread id, the starting address, type of operation and extra global status information. Figure 1 shows the format of each entry in AIG. The global status information is all threads running within the program and their corresponding instruction count at the moment the memory reference happens. This information is used for chunking. For example, when two shared memory accesses need to be ordered, successor operation in thread A triggers chunk termination of block containing previous operation in thread B. Instruction count for thread B is required here to compute precise chunk size.

We develop our own pin-tool to construct AIG. The main algorithm is in Algorithm 1.
Algorithm 1: AIG Construction Algorithm

Data: Program Execution
Result: AIG

1 Function Instrument at thread creation
2 if Thread Created then
3 map thread id to readable integer;
4 clear instruction count;
5 end
6 end

7 Function Instrument at memory write/read
8 if Memory ReadWrite then
9 align starting and end address according to unit_size;
10 while Exist address unit not processed do
11 record current thread id, starting address, type of operation, instruction count;
12 end
13 get scope lock;
14 iterate global status data structure, print out available threads information;
15 release scope lock;
16 end
17 end

4.2 Implementation

The unit_size in line 6 specifies granularity of address access detection, which can be configured by user with command line “-unit_size #”. In our experiments, we set it the same as cache line size. Such a choice results in chunks terminations due to false sharing. However, this is acceptable since our we assume an x86 basic line architecture with normal cache coherence mechanism (MESI).

We have an elegant design for scope lock in line 10 and 12. As PIN [3] has multiple levels of locks and it leads to deadlock quite easily, programmers have to pay a lot of attention to lock design and maintenance. Our method helps to relieve such concerns significantly. We design a class named as “ScopedLock”, which is constructed by creating a lock with a passed in integer, and deconstructed by releasing and deleting that lock. Programmers should use ScopedLock as a local variable, so that whenever we leave a scope, which is defined by { } including function bounds, the lock is guaranteed to be released. With this, one need not worry about forgetting to release a lock and spending tedious time debugging deadlock caused by that.

<table>
<thead>
<tr>
<th>Num of threads</th>
<th>Ratio of shared access</th>
<th>Shared by # threads</th>
<th>Accessed by # Write/Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>49.4471 %</td>
<td>2</td>
<td>4.99962</td>
</tr>
<tr>
<td>4</td>
<td>72.8668 %</td>
<td>2.00788</td>
<td>5.01403</td>
</tr>
<tr>
<td>8</td>
<td>82.1506 %</td>
<td>2.02428</td>
<td>5.04828</td>
</tr>
</tbody>
</table>

Table 1

4.3 Optimization

As AIG contains all detailed memory access information, it inevitably generates a huge log. One optimization that can be applied is to only record shared memory accesses. Table 1 shows summarized shared address information collected for RADIX. Compared to non-shared memory address, which may
be frequently read and written locally, shared variables has lower write-read times and accessed by fewer distinctive instructions. Therefore, collecting only the shared memory reference helps reducing the log size a lot.

This optimization increases the time to generate logs and consumes high memory usage to remember whether the address has been accessed before. An alternative way is to start recording only after the second thread is created. In our experiments, we see a significant reduction in log size due to this simple optimization. For example, the FFT benchmark in splash2, this modification decrease log size from 608MB to 105KB. The reason for the huge decrease may be that the main thread takes the responsibility to allocate and initialize all arrays, which in general is very big, leading to amount of redundant information.

---

Algorithm 2: Maximum Precision Chunking

1. **Function** *CreateOrderedAccessList*(AIG)
   2. **while** entry E in AIG **do**
   3.     create new Access A from E;
   4.     insert Access A into OrderedAccessList;
   5. **end**
   6. **end**

7. **Function** *PorcessOrderedAccessList*(OrderedAccessList)
   8. **while** Access A in OrderedAccessList **do**
   9.     if A is a Write operation **then**
   10.        insert thread id and accessed address of A to ThreadWriteSet;
   11.     else
   12.        insert thread id and accessed address of A to ThreadReadSet;
   13. **end**
   14. // Increase the chunk size of Access’s Thread
   15. add Access A to Current Thread Chunk;
   16. // Check whether this access causes other threads chunk termination
   17. **while** each thread T **do**
   18.     skip current thread;
   19.     if Access A is in T’s ThreadWriteSet **then**
   20.         ChunkThread(T, A);
   21. **end**
   22. // if A is Write, check other threads’ read set
   23.     if Access A is write operation & A is in T’s ThreadReadSet **then**
   24.         ChunkThread(T, A);
   25. **end**
   26. **end**
   27. **end**

28. **Function** ChunkThread(Thread, Access)
29.     static GlobalTimeStampCounter;
30.     // used to order chunks
31.     get instruction count C of thread T from Access A;
32.     create new chunk with C and reset chunk size of thread T;
33. **end**

34. **Function** *EstimateSerialReplay*(ChunkLog)
35. **while** Log Entry exists **do**
36.     Total Cycle += Chunk Manage Overhead + Instrumentation Overhead;
37.     Total Cycle += Chunk Size × IPC;
38. **end**
39. **end**
5 Max-Precision

The objective of this code is to model the serial replay of processed QuickRec [5] replay logs. This
is done by first building an Ordered Access List of all the shared memory accesses that occur during
the programs execution. This data is provided by our AIG tool. The pseudo code is provided in
CreateOrderedAccessList(). The next step is to emulate the chunking mechanism of the QuickRec
hardware platform but without any of the imprecisions of the QuickRec platform. The code for this is in
ProcessOrderedAccessList(), which consumes the OrderedAccessList to create the globally ordered
chunk log. Finally the code in EstimateSerialReplay will consume the ChunkLog to compute a total
cycle count as a function of ChunkManagement and InstrumentationCosts.

6 Max-Replay

6.1 Algorithm Overview

<table>
<thead>
<tr>
<th>Algorithm 3: Token-Based Replay Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Function processMemRef()</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>4 Function callTokenBasedChunking()</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11 end</td>
</tr>
<tr>
<td>12 Function terminateChunkAfter(memory reference)</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16 end</td>
</tr>
<tr>
<td>17 Function stitchChunks()</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23 end</td>
</tr>
<tr>
<td>24 Function doPerformanceModeling()</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>28 end</td>
</tr>
<tr>
<td>29 Function main()</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34 end</td>
</tr>
</tbody>
</table>

6.2 Code Description

Here’s the description for some basic functions in Algorithm 3.

processMemRef()

This function goes over all the memory references (essentially a bundle of the memory addresses and
the thread-id requesting the access) in the AIG log, and creates predecessor-successor relations. For each reference, it is appended to the list of the corresponding memory address it accesses. Thereafter, each memory address gets a list with all the references attached to it. If the memory reference A to address X is a write, then the last memory reference to X from each other thread is added as a predecessor of A. Also, for the memory references added as a predecessor, the current memory reference A is added as a successor. However, there are two special conditions we have exclude. One is that if the memory reference is a read, then appropriate steps are taken to not count for R→R as predecessor-successor dependence. The other is that, within the same thread, dependencies are not included since they are implicitly ordered while doing a replay.

callTokenBasedChunking()

This is where we determine the chunking bound to achieve maximal replay parallelism. We go over all the memory references in the AIG log. If the memory reference has a predecessor, then the chunk is terminated right before the memory reference. If the memory reference has a successor, then the chunk is terminated right after the memory reference (i.e. including the memory reference within the terminated chunk).

terminateChunkAfter() / terminateChunkBefore()

These functions create the actual chunks. terminateChunkBefore() reduces the waste in the destination chunk. In Figure 2b, waste is the bubble of instructions from the start of chunk to just before the destination memory reference (the empty block above “W”), which would otherwise have been executed sequentially after source chunk (containing “R”), due to the dependence introduced by destination instruction (“W” in Figure 2b). terminateChunkAfter() reduces the waste in the source chunk (containing “R”). Waste is the bubble of instructions from the source memory reference to the end of the chunk (the empty block after “R”). When the new chunks are created, a critical step is set them up to receive tokens from previously created chunks.

6.3 Token Passing

Since chunks are only created at memory reference boundaries, the ordering of the newly created chunk is determined to execute after the predecessors of the memory reference completes. For each reference, the corresponding chunk of the predecessor is set up for passing a token to the newly created chunk (in Figure 2b, chunk containing “R” is set up for passing a token to the chunk with “W” inside). A token is passed from one chunk to another when the former chunk completes execution of all the instructions in that chunk. Logically, the newly created chunk is set up for receiving a token from the predecessor chunk. In a general scenario, a chunk may be set up for receiving N tokens, and it can’t start executing before all N tokens have been received.

6.4 Stitching Chunks

The replay algorithm in Algorithm 3 gives us the theoretical bound for maximum replay on a parallel machine. Best performance can be observed when chunk management overhead from Pin is negligible.
Chunk management overhead is defined as the cycles consumed by Pin to schedule the next chunk for a particular core. More specifically, a priority queue of chunks is maintained in Pin. This overhead would involve removing the next chunk from the queue and starting execution. However, if the chunk management overhead is non-negligible, the maximum replay algorithm could potentially hurt replay speed due to the large number of chunks created by this algorithm to achieve finer-grained parallelism. Consider the following case in Figure 2c. The instruction writing to memory is ordered with respect to a predecessor read, and also with respect to a successor read. It is possible that, in such a case, our algorithm will create a single-instruction chunk of “W”. In the worst case scenario, the maximum replay algorithm could create a separate chunk for each memory reference in the program thereby leading to chunk explosion, and with a non-negligible chunk management overhead, the performance would surely tank.

According to this observation, we extended our algorithm to stitch some of the chunks created by the maximum replay algorithm. Stitching refers to combining two or more consecutive chunks from the same thread. If chunks A and B are stitched to form chunk C, then the total instruction count in chunk C is the cumulation of the instruction count of chunk A and B. Also, the token passing is adjusted, otherwise stitching would not have taken effect. All the predecessors of A and B are made the predecessors of C, and all the successors of A and B are made successors of C. An example stitching is shown in Figure 2d. In our algorithm, we keep the maximum number of chunks which can be stitched together as a configurable parameter, and do a design space exploration to find the optimal number of chunks to stitch for different chunk management overheads (reflective of different machine hardware on which replay is being performed). This is summarized in section 7.

If the optimal number of chunks which should be stitched is say N, it may or may not be possible to actually find N consecutive chunks from a thread to stitch together. This is because chunks can be combined only if they do not introduce cycles in the chunk dependence graph. A cycle can be created when a chunk A (in thread 1) is waiting for tokens from chunk B (in thread 2), and at the same time chunk B is waiting for tokens from chunk A or any chunk after A in the same thread. This scenario causes a deadlock. To check for existence of such deadlocks, our algorithm uses Lamport Scalar Clocks [2] in a way similar to the idea in [1]. Specifically, each chunk is assigned a clock (scalar value) that is strictly larger that all of its predecessors. For an existing stitched chunk A, a new chunk B is stitched to A only if all the predecessors of B have the Lamport clock number not larger than that of the first chunk in A. With this condition, no cycles are created and we can safely merge B. If, however, one of the predecessors of B violates this condition, we stop merging and start a new stitching.

In summary, high degree of stitching may be desired when the chunk management overhead for replay is high on a system, and also when the space overhead to store the chunk logs is a concern.

7 Performance Estimation

We wrote a simple multi-core performance model to evaluate the cycles taken to replay the AIG logs using maximum replay algorithm, and also the change in performance when applying stitching. The performance model takes the average CPI of the program and chunk manage overhead as input. We use the perf tool on linux to get the average CPI for an application by running it directly on hardware, and feed that number to our performance model. And the chunk management overhead is a configurable constant.

The performance model simulates a 4-core machine. Each core has a queue from which chunks are scheduled whenever the core is idle. In case the next chunk to be run on the core has not received the necessary tokens, we make the core spin.

8 Evaluation and Results

In this section we summarize and analyze our results. Figure 3 shows for 10 benchmarks from the SPLASH suite, the speed-up of the max-precision algorithm and the max-replay algorithm over the execution in the QuickRec implementation. For max-precision and max-replay numbers, we add a chunk management overhead (in Pin replayer) of constant 500 cycles. This overhead is the time taken by Pin to claim the next chunk from the priority queue of the core, and schedule it for running.

On average, our results suggest that max-precision improves performance over QuickRec, albeit by a small amount (average = 0.4%). The reason for the improvement not being significant is as follows - Max-precision reduces the total number of chunks as compared to QuickRec drastically by eliminating
chunks created by false dependencies. However, since the replay is still serial, and the chunk management overhead is small (only 500 cycles), we do not see a large improvement in the final absolute cycle count. It should be noted that on a system where the chunk management overhead is appreciable, our results would look better (as shown in Figure 4).

<table>
<thead>
<tr>
<th>Application</th>
<th>Average Chunk Size for Max-Replay</th>
<th>Average Chunk Size for Max-Precision</th>
<th>Ratio (Max-Precision chunksize / Max-Replay chunksize)</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes</td>
<td>4779</td>
<td>627251</td>
<td>131</td>
</tr>
<tr>
<td>radix</td>
<td>10000</td>
<td>23932</td>
<td>2</td>
</tr>
<tr>
<td>fmm</td>
<td>28969</td>
<td>374208</td>
<td>13</td>
</tr>
<tr>
<td>water-nsq</td>
<td>163636</td>
<td>5157787</td>
<td>32</td>
</tr>
<tr>
<td>ocean-nc</td>
<td>655581</td>
<td>1326775</td>
<td>2</td>
</tr>
<tr>
<td>raytrace</td>
<td>1647259</td>
<td>5339293</td>
<td>3</td>
</tr>
<tr>
<td>lu</td>
<td>3847556</td>
<td>16478163</td>
<td>4</td>
</tr>
<tr>
<td>ocean</td>
<td>10921250</td>
<td>43912484</td>
<td>4</td>
</tr>
<tr>
<td>lu-nc</td>
<td>4117647</td>
<td>11274558</td>
<td>3</td>
</tr>
<tr>
<td>fft</td>
<td>5463092</td>
<td>17661685</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2: Chunk Size
The problem of serial replay is solved by the max-replay algorithm, which parallelizes the replay. The speedup over QuickRec is shown by the blue bars. On average across SPLASH, we see an improvement of 3.7x in performance with 4 cores. Also, note that this is the theoretical bound for maximum parallel replay performance. This is because the replay algorithm aggressively removes any unnecessary ordering of instructions in the chunks by breaking chunks into smaller pieces, and in the process creating the maximum number of total final chunks possible. Table 2 shows the average chunk size (number of instructions in the chunk) for the max-precision and the max-replay algorithms. As expected, the average chunk size for the latter is usually an order of magnitude smaller.

In Figure 5, we present an interesting case of a different version of Radix (not from the SPLASH suite). This version seems to have a lot more sharing, and hence chunk terminations due to RAW, WAR and WAW. The average chunk size for this version was found out to be only 43 instructions. We decided to evaluate this case further as it presents a scenario where the max-replay algorithm may not hold to be the theoretical bound for maximum replay performance (unlike SPLASH). The reason for this is enumerated in the graph which shows the performance vs. stitching bound. Stitching bound is defined as the maximum number of chunks that are allowed to stitch together. Consider the case of our usual chunk management overhead (=500 cycles, green line). As can be seen, for the max-replay case (stitching bound=1), since the number of chunks is huge (approx. 50000) and the average number of instructions per chunk is low (approx. 43), this management overhead contributes a lot towards the overall performance. This is unlike SPLASH where the number of chunks was low, and the average instruction count per chunk was decent. For the green line, the minima (best performance) is observed at stitching bound value of 21. We summarize our findings with the statement that the max-replay algorithm may not present a theoretical bound to maximum replay performance if 1) there are a large number of very small (instruction-count-wise) chunks 2) the chunk management overhead is high.

References


