Rainbow: Efficient Memory Dependence Recording with High Replay Parallelism for Relaxed Memory Model *

Xuehai Qian*, He Huang◇, Benjamin Sahelices†, and Depei Qian‡

* University of Illinois Urbana-Champaign ◇ AMD Products (China) Co., Ltd.
† Universidad de Valladolid ‡ Beihang University
xqian2@illinois.edu henry.huang@amd.com benja@infor.uva.es depeiq@buaa.edu.cn

Abstract

Architectures for record-and-replay (R&R) of multithreaded applications ease program debugging, intrusion analysis and fault-tolerance. Among the large body of previous works, Strata enables efficient memory dependence recording with little hardware overhead and can be applied smoothly to snoopy protocols. However, Strata records imprecise happens-before relations and assumes Sequential Consistency (SC) machines that execute memory operations in order.

This paper proposes Rainbow, which is based on Strata but records near-precise happens-before relations, reducing the number of logs and increasing the replay parallelism. More importantly, it is the first R&R scheme that supports any relaxed memory consistency model. These improvements are achieved by two key techniques: (1) To compact logs, we propose expandable spectrum (the region between two logs). It allows younger non-conflict memory operations to be moved into older spectrum, increasing the chance of reusing existing logs. (2) To identify the overlapped and incompatible spectra due to reordered memory operations, we propose an SC violation detection mechanism based on the existing logs and the extra information can be recorded to reproduce the violations when they occur. Our simulation results with 10 SPLASH-2 benchmarks show that Rainbow reduces the log size by 26.6% and improves replay speed by 26.8% compared to Strata. The SC violations are few but do exist in the applications evaluated.

1. Introduction

The trend towards multi-core processors has spurred a renewed interest in developing parallel programs. The current non-deterministic shared-memory multiprocessors exhibit different behaviors across runs. Record-and-Replay (R&R) has broad uses in at least program debugging [1, 15, 25], where a concurrency bug can be reproduced; intrusion analysis [10, 14], where an intrusion can be traced back to an attacker's actions; and fault-tolerant, highly-available systems [9, 24], where a backup machine can resume from where the primary failed.

Two-phase deterministic R&R [4, 7, 13, 18, 20, 22, 23, 27, 28, 29] seeks to monitor the execution of a multithreaded application on a parallel machine, and then reproduce it exactly later. It involves recording in a log all the non-deterministic events that occurred during the initial execution (i.e. application inputs and memory operation interleavings). Then, during the replay, the recorded inputs are provided to the application at right times, and the memory operations are forced to have the same interleavings as the original execution.

While the execution events (e.g. context switch and I/O, etc.) can be easily logged by the software [10, 16, 19], *memory race recording* remains a challenge. Early approaches record memory races when threads *do* communicate [28, 29], they require substantial hardware overhead to restrict log size. Recent approaches alternatively record the periods that threads *do not* communicate. The sequence of communication-free dynamic instructions from a single thread is called a *episode* [4, 13] or *chapter* [27] or *chunk* [18].

Among different approaches, Rerun [13] operates on conventional point-to-point directory protocol. Karma [4] optimizes the serial replay of Rerun by recording the order of episodes by directed acyclic graph (DAG). It can also reduce the log size with delayed episode termination on cycle. Timetraveler [27] reduces the log size by post-dating. Typically, reduced log size makes replay slower [4], because each episode (or chapter) covers more dynamic instructions and therefore larger regions of instructions are serialized on dependence. DeLorean [18] is based on BulkSC [6], a non-conventional architecture that continuously executes and commits implicit atomic blocks (chunks). Thanks to the semantics of BulkSC, Delorean naturally enables parallel replay, as long as chunks are committed in the same order as in the original run.

The above approaches either target directory-based design or require special coherence protocol. In this paper, we are interested in R&R schemes that are easily applicable to the snoopy protocol, which is by far the most commonly used design.

Three major R&R schemes suitable for snoopy protocols are Strata [20], Intel Memory Race Recorder (IMRR) [22] and CoreRacer [23]. A Strata log is a vector of as many counters as processors. It creates a time layer across all the running threads, which separates all the memory operations executed before and after. A single stratum can potentially capture many dependences. More importantly, it naturally enables the parallel replay of all the memory operations in a strata region. IMRR [22] is a chunk-based approach targeting replay speed. The key idea is to terminate the chunks synchronously and place the destination of a dependence in the next chunk. This technique generates independent chunks with the same Lamport Clock (LC) that can be replayed in parallel. In fact,

*This work was supported in part by the National Science Foundation of China under grants 61073011 and 61133004, Spanish Gov. & European ERDF under TIN2007-66423, TIN2010-21291-C02-01 and TIN2007-60625, Aragon Gov. & European ESF under gaZ: T48 research group, Consolider CSD2007-00050, and HiPEAC-2 NoE (European FP7/ICT 217068).

the IMRR logs generated in this way surprisingly resemble the strata log, — all the processors need to terminate the current chunk and record the size of the previous chunk on a dependence between any two processors. It confirms the merit of parallel replay in Strata: for higher replay speed, chunk-based approaches need to create logs in a similar manner. CoreRacer [23] focuses on supporting TSO memory model for chunk-based schemes.

This paper proposes *Rainbow*, a R&R scheme that reduces log size and increases replay speed at the same time. Rainbow records near-precise happens-before relations. A log in Rainbow is called an *arch*, which is similar to a strata log except that the counters for each thread are adjustable. The region between two arches is called a *spectrum*, which is similar to a strata region except that it can be expanded. *Expandable spectrum* is the key technique that enables the near-precise happens-before relation recording. The insight is, after the creation of an arch, which separates the previous and current spectrum, the future memory operations in the current spectrum can be *moved* to the previous spectrum if there is no conflict and the program order within the local thread is preserved. This technique seeks to place more conflict-free memory operations into the existing spectra. As the memory operations are moved to the older spectra, the existing arches have more chance to be reused to capture new dependences and the spectra tend to have more conflict-free operations that can be replayed concurrently. Indeed, Rainbow targets the ideal scenario, — *creating a new arch only when the existing arches cannot capture a new dependence*.

Rainbow is also the first R&R scheme that supports *any* memory consistency model. Strata works well with the SC implementation where the memory operations are executed in order. It is unclear how Strata works with an SC implementation with in-window speculation and out-of-order execution [12], because *the exposure of coherence transactions and dependences can be inconsistent with the sequential execution order*. The out-of-order execution and relaxed memory model can cause the overlapped and incompatible spectra that cannot be replayed. Special care must be taken to record the arches correctly. The key to support relaxed memory model in R&R is to detect and record SC violations and faithfully reproduce the same interleaving in the replay. We propose a novel SC violation detection method based on the existing arches. To reproduce the execution, the arches conditionally record extra information (the delay and pending set and the values of the SC-violating loads) when SC is violated.

This paper makes three key contributions:

- Expandable spectrum: enabling the near-precise recording of happens-before relations. It reduces the log size and increases the replay parallelism simultaneously.
- SC violation detection based on existing arch information.
- The mechanisms to record the extra information to reproduce SC violation in the replay.

Overall, Rainbow is the first R&R that supports any relaxed memory model with high replay parallelism.

Using simulations, we show that, on average, Rainbow reduces the log size compared to Strata by 26.6% and increases the replay speed by 26.8%. The SC violations are few but do exist in the applications evaluated.

This paper is organized as follows: Section 2 reviews the previous work with the emphasis on Strata; Section 3 explains expandable spectrum; Section 4 supports relaxed memory model in Rainbow. Section 5 discusses the implementation issues. Section 6 evaluates the proposed techniques. Section 7 concludes the paper.

2. Prior Work and Strata Review

2.1. Background on Deterministic R&R

Deterministic Record-and-Replay (R&R) consists of monitoring the execution of a multithreaded application on a parallel machine, and then exactly reproducing the execution later. R&R requires recording in a log all the non-deterministic events that occurred during the initial execution. They include the inputs to the execution (e.g., return values from system calls) and the order of the inter-thread communications (e.g., the interleaving of the inter-thread data dependences). Then, during replay, the logged inputs are fed back to the execution at the correct times, and the memory accesses are forced to interleave according to the log. Deterministic R&R has been the subject of a vast body of research work.

2.1.1. Software-based Approaches Software-based solutions [5, 8, 10, 11, 24, 25] assume no additional hardware support and rely on modified runtime libraries, compilers, operating systems and virtual-machine monitors to capture sources of non-determinism.

Early approaches either are inherently designed for uniprocessors or suffer significant slowdown when applied to multiprocessor executions. DoublePlay [26] is an effort to make replay on commodity multiprocessors more efficient. It timeslices multiple threads on one processor and then runs multiple time intervals on separate processors. ODR [2] and PRES [21] are probabilistic replay techniques for reproducing concurrency *bugs*. The idea is to record only a *subset* of non-deterministic events required for deterministic replay and use a replayer that searches the space of possible executions to reproduce the same application output or bug, respectively. Respec [17] records a subset of non-deterministic events and uses the online replay to provide external determinism.

2.1.2. Hardware-assisted Memory Race Recording Hardware is mainly used in memory race recording to reduce the overhead of R&R for multiprocessor executions. Bacon and Goldstein [3] pioneered the idea of recording memory races in hardware. Races are recorded by logging coherence messages on the snoopy bus, producing a serial and voluminous log.

FDR [28] and RTR [29] record dependences between pairs of instructions to get parallel dependence graphs, still incuring large logs and overhead. To reduce this overhead, chunk-based techniques [4, 7, 13, 18, 22, 23, 27] have been proposed. They record the periods that threads do not communicate.

However, most of them are not designed for parallel replay. DeLorean [18] allows parallel replay but is based on a non-conventional BulkSC architecture.

Karma [4] is the first chunk-based R&R technique that explicitly targets replay parallelism on conventional architecture. It improves the replay speed by eliminating artificial restriction, — replacing the scalar timestamps by DAG. However, it doesn't remove the fundamental requirement: two episodes with dependences still have to be replayed in serial order. Therefore, the excessive large episode makes replay slower, since larger region of dynamic instructions are serialized. In the example in Figure 1 (a), the four episodes are required to be replayed in serial order. Such scenario is not possible in Rerun [13] because the source of a dependence has to terminate the current episode, but Karma [4] (and Timetraveler [27]) removes the requirement and achieves smaller log size by having larger episodes (chapters). The key observation is that, in a chunk-based solution, log size and replay speed are usually opposite optimization goals. Each serialization can only resolve one or more dependences between two episodes (chapters) in the original execution.

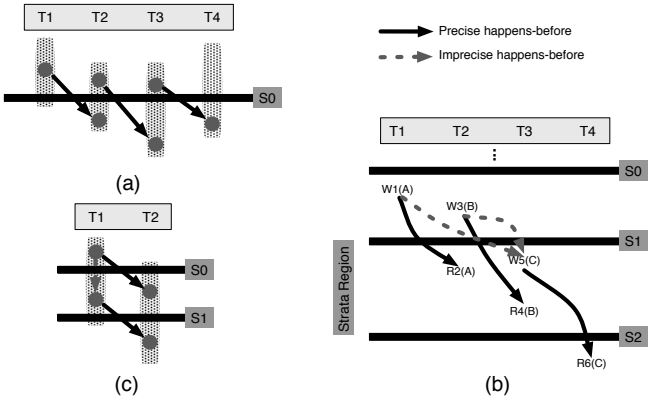

Figure 1: Review of Strata and Chunk-based R&R Schemes

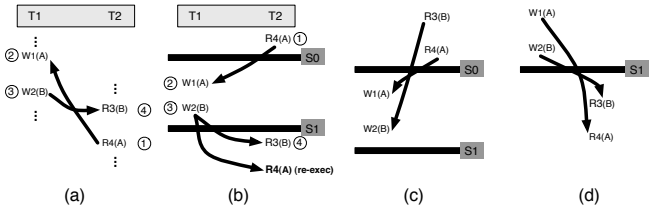

Figure 2: Implication of out-of-order execution on Strata

2.2. Strata Review

Compared to the directory-based memory race recording solutions, schemes suitable for snoopy protocols are fewer and the three major recent proposals are Strata [20], IMRR [22] and CoreRacer [23]. In the following, we discuss Strata in detail since Rainbow is based on it.

Strata [20] requires that all the processors agree to start a new strata region when there is a dependence between any two processors. This is done by augmenting the messages with a "Log Stratum" bit, which can be set by the processor initiating the miss or by a processor that provides the data. Strata uses a recording approach that requires that all processors record an entry in their logs at the same time.

Except the limited scalability, Strata incurs little hardware and bandwidth overhead. It has the potential to achieve small log size: although each strata log is a vector of counters, it may capture many dependences that, in point-to-point or chunk-based solutions, require more logs to record. Moreover, Strata ensures good replay speed, since all the memory operations in a strata region can be naturally replayed concurrently. A single serialization can enforce multiple dependences. For the example in Figure 1 (a), Strata only incurs one strata log. During replay, the order enforced between two strata regions by the strata log is sufficient to enforce the correct order of all three dependences. Finally, in Strata, the smaller log size and higher replay speed are *compatible* optimization goals. Fewer strata logs imply larger strata regions, which translate to higher replay speed. Due to these merits, we believe that Strata is the superior choice for snoopy protocol.

2.3. Problems with Strata

2.3.1. Imprecise happens-before relation recording To reuse the current strata log to capture more dependences, Strata distinguishes the memory operations in the current and previous strata region. In the following examples, W(A) (R(A)) indicates a write (read) to address A. In Figure 1 (b), S1 is created due to a RAW: W1 → R2. After S1 is created, W1 and W3 are in previous strata region and R2 is in current strata region. Later, when R4 is performed in T_3, no new strata log is needed, because the new RAW is from a write (W3) in previous strata region to the current one, and S1 already captures the dependence. Suppose there is another write W5 to address C in T_3 after S1, when T_4 reads from address C, a new strata log S2, is created, because the RAW: W5 → R6 cannot be captured by any existing strata log.

It is easy to see that a strata log may enforce some *imprecise* relations as *side effect*. We mark the precise and imprecise happens-before relations in Figure 1 (b) as solid and dashed arrows, respectively. More specifically, by creating S1, two imprecise happens-before relations are enforced, they are W1 → W5 and W3 → W5. Since W1, W3 and W5 are conflict-free memory operations, ideally, there should not be any order among them. Such imprecise happens-before relations incur more strata logs and can limit replay parallelism.

It is worth noting that Strata can also incur unnecessary serialization between memory operations from the same thread. The problem is shown in Figure 1 (c). Suppose the two dependences are on different addresses. The chunk-based approaches (e.g. Karma [4] or Timetraveler [27]) don't need to terminate episodes (or chapters) in both threads, since they leverage the conflict serialization. However, Strata still needs to create two strata logs, S0 and S1. In the replay, the second operation in T1 has to wait for all the memory operations before S0 is completed. The key problem is that S0 enforces an "*implicit fence*" between the two memory operations in T1, which is not necessary (supposing that in the original code there is no fence between them).

2.3.2. Interference of Out-Of-Order Execution Strata assumes an SC implementation that executes memory operations in order. For a more aggressive SC implementation, the out-of-order execution and in-window speculation may interfere with the creation of strata logs. It is unclear how Strata handles these scenarios, because the cache accesses and coherence transactions do not always appear to be in the same order as the commit order of the instructions.

The issue can be seen in Figure 2 (a). In an aggressive SC implementation, R4 in T_2 can execute before R3 (①), and then W1 and W2 in T_1 are retired from processor and globally performed in order (② → ③). Since R4 and W1 are conflict accesses, a WAR: R4 → W1 is formed. Finally R3 in T_2 is executed (④), it reads the value produced by W2, a RAW: W2 → R3 is formed. At this point, the processor of T_2 will detect an SC violation, since a pending load R4's address B is invalidated before it retires from the processor and after it reads the value. The processor will re-execute from the SC-violating load R4 to recover correct SC semantic. During the re-execution, R4 will first miss in private cache and then get the data produced by W1. This incurs another RAW: W1 → R4. From this example, we see that the coherence transactions and dependences may appear in different order as the commit of the instructions, and extra temporary dependences can be established due to SC violations.

Figure 2 (b) shows how Strata works in the example. For a better understanding, we assume all three kinds of dependences are recorded. The first WAR and second RAW create S0 and S1, respectively. The third RAW due to the re-execution of R4 can be captured by S1, therefore, no extra dependence is created. Unfortunately, S0 and S1 are contradicting: S0 requires that T_2 can execute up to R4 and W1 in T_1 cannot execute until all memory operations before S0 are completed. This leads to the replay order in Figure 2 (c). It is incorrect and the interleaving is different from the original execution. In fact, Strata does record the necessary (and correct) strata log that can lead to correct replay, it is S1. However, after finishing S0 in Figure 2 (c), it is impossible to enforce the requirement of S1: R3, which should get the value from W2, has already executed. It is because R3 is in program order before R4.

After the re-execution of SC-violating memory operations, the order of the four memory operations is: ② → ③ → ④ → ①. If the system would have ignored S0 and only used S1, the execution can be correctly replayed, as shown in Figure 2 (d). Strata lacks the proper mechanism to detect and fix the contradicting (and incorrect) strata logs (e.g. S0).

The example makes clear that special care must be taken for the SC aggressive implementations. To accommodate the out-of-order execution effects, the complete scheme should detect the transient SC violation cases and compensate the effects in replay. These violations are *transient* since the SC hardware will detect them and re-execute problematic instructions eventually. For a machine with relaxed memory model, similar techniques can be also applied. However, the difference is that, the hardware will *not* recover the SC semantic. The record phase has to record necessary extra information conditionally in a log, when the out-of-order effects are visible and cause SC violations.

In the following sections, we propose two novel techniques to resolve the problems.

3. Near-Precise Happens-before Recording

In this section, we explain the technique to eliminate most of the imprecise happens-before relations in Strata. We do not assume aggressive SC implementation, the issues with it will be addressed in Section 4.

3.1. Rainbow Insight: Expandable Spectrum

The imprecise happens-before relations in Strata are due to two reasons. First, the strata log is a global notion, which partitions the previous and current strata region in *all* threads. Second, after a strata log is created, it closes and finalizes the previous strata region. All the future memory operations have to be included in the current region. In Rainbow, the format of the log, called *arch*, is the same as Strata. The region between two arches is called a *spectrum* (plural *spectra*). Rainbow seeks to achieve the following property.

3.1.1. Desired Rainbow Property Any two memory operations from two threads (T_i:A and T_j:B) are divided by an arch if and only if one of the following condition holds:

(a) There is a dependence between them: A $\xrightarrow{dep}$ B.

(b) There is a dependence between A and B', a memory operation before B in T_j in program order: A $\xrightarrow{dep}$ B' and B' $\xrightarrow{po}$ B.

The property ensures that only the precise happens-before relations are recorded. To achieve it, Rainbow introduces the concept of *Expandable Spectrum*. The insight is simple: for memory operations that do not conflict with the previous spectrum, they can be *moved* to the previous spectrum as long as the program order is preserved.

3.1.2. Spectrum Expansion Rules A memory operation A from T_i can be moved to the previous spectrum if both of the following conditions hold:

(a) In the current spectrum, there is no memory operations in T_i.

(b) In the previous spectrum, there is no conflict operations in threads other than T_i.

Condition (a) ensures the program order of memory operations in the local thread is preserved. Condition (b) ensures that all the dependences are captured by at least one arch.

Moving memory operations to the previous spectrum is implemented simply by increasing the memory count of a thread in the existing arch. In Strata, once the strata log is recorded, the counts in it are unchangeable. In Rainbow, all the counts in an arch are adjustable, until a previous spectrum is closed. We show the condition for closing a spectrum soon.

For each spectrum, we need to keep the read and write address set for each thread. To reduce the hardware cost,

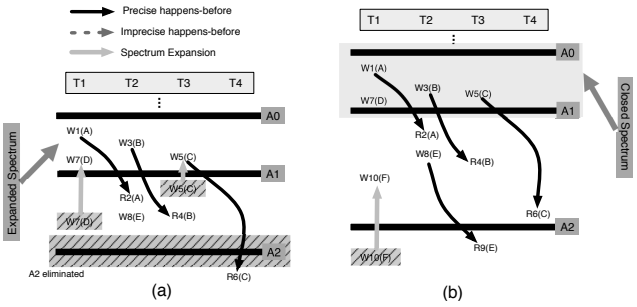

Figure 3: Expandable Spectrum

Rainbow keeps a combined write and read set of all threads for each spectrum, instead of per-thread sets. This approximation is part of the reason why Rainbow is near-precise.

3.1.3. Spectrum Expansion Examples Figure 3 (a) shows how spectrum expansion works for the same example in Figure 1 (b). The arches are denoted as Ai. After A1 is created due to W1 → R2, W5 in T_3 is performed. Since there is no memory operation from T_3 in the current spectrum and there is no conflict memory operation in the previous spectrum, W5 is moved up before A1. Later, when R4 is performed in T_3, we cannot move it to the previous spectrum, since there is a conflict write W3 in T_2. No new arch is created since A1 captures this dependence. After R4 is performed, the future memory operations from T_3 are not eligible to move to the previous spectrum. It is because any move will reorder the later memory operations beyond R4. Similarly, W8 in T_2 cannot be moved. When T_1 performs W7 after A1, it can be similarly moved to the previous spectrum.

Finally, when R6 from T_4 is performed, *no new arch* is created, since W5, the source of the dependence, has been moved to the previous spectrum. The dependence W5 → R6 can be captured by the existing arch A1.

Figure 3 (b) shows the situation when a spectrum is closed, assuming the system maintains the state of two most recent spectra. More generalized case is discussed in Section 3.2. Following the previous example, R9 in T_3 incurs a new arch A2, it forms a new RAW dependence with W8. After A2, the spectrum between A1 and A2 becomes the "previous" region and the current spectrum starts from A2. At this point, the spectrum between A0 and A1 is *closed*, — not expandable any more. At this point, the current log for the closed spectrum is recorded and can never be changed. Closing a spectrum means no memory operation can be moved into it. Therefore, when a new write W10 in T_0 is performed, it can be at most moved to the spectrum between A1 and A2.

3.1.4. Merits of Spectrum Expansion In the example of Figure 3 (a), we see that Rainbow records one new arch (A1) while Strata records two (S1 and S2). The spectrum between A0 and A1 is expanded to contain two extra memory operations: W7 and W5. A2 is eliminated because the existing A1 can be *reused* to capture new dependences after the previous spectrum is expanded. Spectrum expansion increases the chance that an existing arch can be reused to capture more dependences.

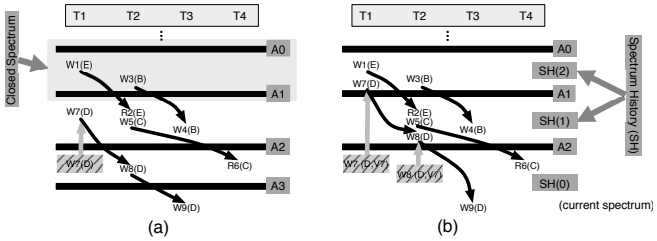

Figure 4: Spectrum History

Rainbow has two merits compared to Strata. First, it reduces the number of logs created, resulting in smaller log size. Second, since each spectrum contains more memory operations, all of them can be replayed in parallel, increasing the replay speed.

3.2. Spectrum History

3.2.1. Motivation Spectrum expansion eliminates the imprecise happens-before relations from the previous to current spectrum. But after the previous spectrum is closed (a new arch is created), some imprecise happens-before relations can still be enforced, which may later incur extra arches.

The problem can be seen in the example shown in Figure 4 (a). A1 is created due to W1 → R2, and later captures W3 → W4. A2 is created due to W5 → R6 and it closes the spectrum between A0 and A1. When W7 in T_1 is performed, it can be moved to the spectrum between A1 and A2. After the expansion, when W8 is performed, no new arch is created since A2 can capture the dependence W7 → W8. When W9 to the same address D is performed in T_3, a new arch A3 needs to be created, since W8 is in the current spectrum and no existing arch can capture W8 → W9.

Ideally, it is correct if W7 were moved further to the spectrum between A0 and A1. It is because T_1 doesn't have any memory operation in the spectrum between A1 and A2 and the address D is not written in the earlier spectrum between A0 and A1 by any other threads. Such scenario is shown in Figure 4 (b). If we allowed such move, later, W8 can be moved to the spectrum between A1 and A2. The difference is that the dependence W7 → W8 is captured by reusing A1, instead of A2. This still satisfies the condition in Section 3.1.2. Such difference is manifested when W9 is performed. In the original case, A3 is created, while in the ideal case, no new arch is needed. Because W8 has been moved to the previous spectrum between A1 and A2, the existing arch A2 can be reused to capture W8 → W9.

3.2.2. Spectrum History Mechanisms Motivated by the example, we propose to have *Spectrum History (SH)* in the record phase. Spectra in SH are considered to be *active*, when a spectrum is closed, it becomes *inactive*. SH keeps multiple active spectra, so that memory operations can move across *multiple* spectra. A memory operation can be moved up until it hits a conflict spectrum, — either some other threads access the same address, or the local thread is not empty in a spectrum.

SH is organized as a FIFO buffer that keeps a certain number of the most recent previous spectra. When a new spectrum

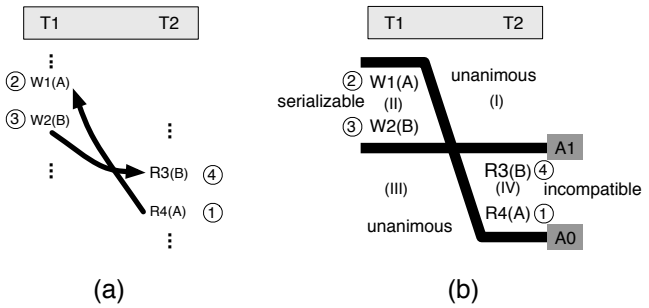

Figure 5: Spectrum with out-of-order execution

is created, all the SH entries are shifted to an older position, so that the current spectrum can take the newest place in the buffer. As a result, the oldest spectrum in SH is closed.

Intuitively, SH seeks to *compact* the memory operations into fewer spectra. With an *unlimited* SH and per-thread address sets, we can get the *optimal* memory operation placement in spectra. In practice, only a limited number of SH entries are kept. The farthest move for a memory operation is from the current spectrum to the oldest spectrum in SH, even if an extra move is still correct. The basic expandable spectrum described in Section 3.1 has a SH size of 1, — it only keeps information of the most recent spectrum. Strata does not have any SH, so the previous region is closed as soon as a new strata log is created. The concept of SH generalizes the Strata and different Rainbow designs (with different SH sizes) into a unified framework.

4. Dependence Recording under Relaxed Memory Model

In this section, we discuss the effects of out-of-order execution. We first try to understand the issues, then propose the solution to handle out-of-order memory operations in a machine with SC and relaxed memory model.

4.1. Understanding Out-Of-Order Effects

We consider the same example in Section 2, shown in Figure 5 (a). In Rainbow, two arches are created. A0 is the first arch, which orders R4 and W1; A1 is the second one and orders W2 and R3. These two arches incur the problem in the replay, because finishing A0 already implies the execution of some instructions *after* A1. The issue is due to the *overlapped spectra*. Figure 5 (b) clearly shows the insight. The two arches partitions the space into four "quadrants", marked (I) to (IV). Among the four quadrants, (I) and (III) are *unanimous*, both A0 and A1 require the same order. For unanimous quadrants, the current logging mechanisms does not cause problem. Quadrant (II) does not cause problem either. Although they need to be ordered after A0 and before A1, such order is *serializable* and can be satisfied in the replay by processing A0 and A1 in order. The quadrant causing problem is (VI), we call it *incompatible* quadrant. The instructions in (VI) are required to execute before A0 and after A1, but A0 is an *older* arch than A1.

From the analysis, we can see the key property to ensure

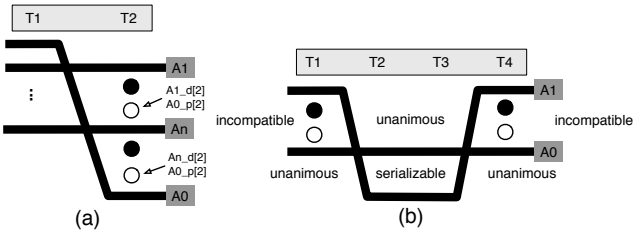

Figure 6: Delay and Pending set

a correct replay: all the incompatible quadrants should be removed. Specifically, *any instruction can only be included in a single spectrum*. Therefore, the problem is to mark the instructions in the incompatible quadrants correctly so that they can be in only one spectrum. We consider two cases: SC with out-of-order execution and Release Consistency (RC). In both machines, the scenario in Figure 5 (b) can happen.

In SC, R4 will be re-executed to recover the SC semantic. Therefore, it should be included in the spectrum after A1 and it cannot be executed before A0. In RC, R4 will not be re-executed and the direction of dependence R4 $\rightarrow$ W1 will never be changed. In this case, R4 should be included in the spectrum before A0.

For the R&R based on Rainbow for relaxed memory models or SC with out-of-order execution, we need to consider two aspects: (1) Record the out-of-order information so that the execution can be replayed faithfully. It requires that each instruction is only in one spectrum. (2) Detecting the cases when such extra information is needed, they are in fact (transient) SC violation from the original execution. We discuss our solutions in the following sections.

4.2. Delay/Pending Sets for Incompatible Spectrum

In Rainbow, we log the the out-of-order effect by introducing two extra sets for each thread in an arch, *delay set* and *pending set*. Because SC violation case like Figure 5 (a) is rare, for most arches, these sets are empty. The extra sets do not increase the log size much. Nevertheless, they are the necessary information needed to be logged if we want to support relaxed memory model.

For T_i and an arch An, the two sets are denoted as *An_d[i]* and *An_p[i]*. The concept is shown in Figure 6 (a). The white circles indicate instructions in the delay set of A1 and An in T_2 (A1_d[2] and An_d[2]), the pending sets of the two arches are empty. The two white circles are also in the pending set of A0 in T_2 (A0_p[2]), A0_d[2] is empty.

The purpose of the two sets is to guide the instructions into the correct spectrum in the replay. Pending set forces the replayer to *skip* some instructions that should be executed later. Specifically, if An_p[i] is not empty, during replay of the spectrum before An, only the instructions *not* in the pending set is executed. Every instruction in the pending set of an arch will be included in the delay set of some other arch. The purpose of delay set is to compensate the previously skipped instructions. Specifically, if the An_d[i] is not empty, after executing the instructions before An from T_i, the instructions in An_d[i] are also executed.

The delay set and pending set are generated as follows. After an arch (A0) is recorded, both sets for each thread are initialized to be empty. Later, we look for a new arch (A1) that can cause the overlapped incompatible spectra, as shown in Figure 6 (b). After identifying such incompatible spectra, the record module dynamically inserts the counts of the instructions that cause the incompatible spectra to the delay set of the new arch and the pending set of the old arch.

Specifically, in Figure 6 (b), the instructions marked with white circle in T_1 are inserted into the A0_p[1] and A1_d[1]. With this extra information, during the replay, when T_1 executes towards A0, it has to skip the instruction in A0_p[1]. Later, after A1 is finished, T_1 has to execute the previously delayed instruction in A1_d[1]. This technique models the reordering effects in the original execution and can be used to faithfully reproduce the same interleaving in the replay. Skipping a store instruction is the same as placing it in the store buffer but sending it to the memory system later. It is not possible to skip a load instruction while retiring the instructions following it. However, the same as re-executing from a load to recover SC semantic, when a load is in a pending set, all the instructions after it are also in the pending set. Therefore, this scenario is not possible.

In the above discussion, we are intentionally vague on the way to detect the incompatible spectra. Such regions are due to the SC violation. In particular, in an SC machine, if the out-of-order effects are repaired by the re-execution, we should *not* record them in the delay and pending set. Detecting the dependence (in opposite direction) due to the re-execution is tricky, since such dependence can be already captured by the existing arch (recall the example in Figure 2 (b). In the next section, we will show how to detect SC violation using the existing arch information and correctly identify the memory operations needed to be inserted in the delay and pending set.

4.3. Finding Incompatible Spectrum Regions

4.3.1. SC Violation Detection and Recording Between Two Processors We use the following simple algorithm to find the incompatible spectra. It essentially detects SC violations based on the existing arches. In Section 4.3.4, we show that the expansion doesn't affect the correctness of our algorithm.

The insight is, during the program execution, if a new dependence is established from a younger spectrum to an older spectrum (divided by an existing arch A0), SC violation occurs. A new arch (A1) is created as normal. The incompatible spectrum is in the thread of the destination of the dependence, from the destination access to the original arch (A0).

Consider the example in Figure 7 (a). First, WAR: R4 $\rightarrow$ W1 incurs an arch (A0). Next, when the RAW: W2 $\rightarrow$ R3 occurs, W2 (source) is in the spectrum after A0 and R3 (destination) is in the spectrum before A0. It indicates an SC violation. After the arch (A1) due to the RAW is created, we can see that it overlaps with the existing A0. The sequence of instructions in T_2 from R3 to R4 (just before A0) compose the incompatible spectrum region, as shown in Figure 7 (b).

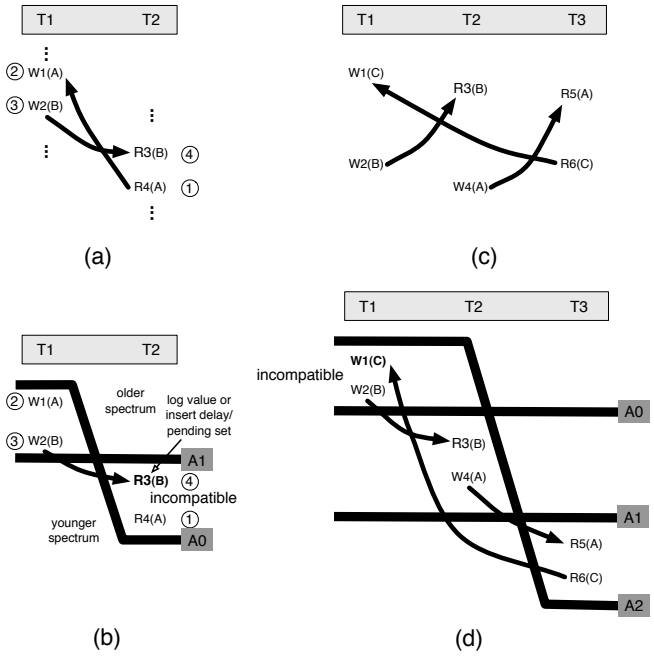

Figure 7: SC violation detection based on spectrum

After identifying the SC violation and the incompatible spectrum, the next step is to record the out-of-order execution effects. If the machine is SC, R4 will be re-executed, and finally forms a RAW: W1 $\rightarrow$ R4. It can be captured by the existing arch A0. If the machine is RC, R4 will keep the old value read before W1 and retire from processor, there is no new RAW dependence. However, whether the new dependence exists will affect the information in delay and pending set. Obviously, R3 needs to be in the pending set of A0 in T_2 (A0_p[2]) and the delay set of A1 in T_2 (A1_d[2]). In SC machine, R4 also needs to be in A0_p[2] and A1_d[2], while in RC machine, R4 should not in these sets. It is because it should indeed be executed before A0 and should *not* be re-executed after A1, otherwise, it will get the new value produced by W1. Specifically, we propose the following mechanism to record the out-of-order effects.

For loads, after identifying the incompatible spectrum region, we log the *values* read by the loads in the region. It is similar to the technique used in previous proposals in handling TSO memory model [4, 29]. We can always log the correct load values, no matter whether the memory model is SC (or whether the extra cache transaction due to the re-execution exists). It is because the values are always logged when a load instruction is retired, if there was a re-execution, then we will log the new value, otherwise, the old value is logged.

If the destination of the dependence from younger to older spectrum is a store, we include that in the pending set and delay set according to the rule in Section 4.2. In the example, if R3 were changed to a store, it will be inserted to the pending set of A0 and delay set of A1.

4.3.2. SC Violation Among More Processors SC violation can happen among multiple processors. In the worst case, it can involve *all* the processors in the system. Rainbow record-

ing scheme must be able to detect them. We show that the spectrum history in Section 3 is not only useful for the log compression, but also the key structure to detect SC violations among multiple threads.

Figure 7 (c) shows an example of SC violation involving three processors. Here, we assume RC machine, so there is no load re-execution to recover SC. First, R3 reads the value produced by W2, then R5 reads the value produced by W4. In T_1, both W1 and W2 have retired from processor and stay in the store buffer when they are trying to perform globally. In this case, W1 is slower, when R6 in T_3 gets the data, W1 is not globally performed yet. Therefore, R6 gets the old value before W1. The SC violation among T_1, T_2 and T_3 happens when W1 is globally performed and is incurred by the last WAR: R6 $\rightarrow$ W1.

Figure 7 (d) shows how the dependences in the above SC violation case are recorded in Rainbow. The RAW: W2 $\rightarrow$ R3 is the first dependence formed. It creates arch A0. Then RAW: W4 $\rightarrow$ R5 is formed and creates the second arch A1. Finally, when the last WAR: R6 $\rightarrow$ W1 forms, the memory count of W1 is smaller than the count for T_1 in A0, while the memory count of the source is larger or equal to the memory count of R6 in T_3, which is guaranteed to be larger than the count for T_3 in A1. It indicates that the last WAR from a younger spectrum to an older spectrum incurs an SC violation. The last WAR creates the third arch A2. We see that the spectrum before A0 in T_1 is the overlapped incompatible spectrum. According to the rule for delay/pending set insertion, W1 is inserted into the pending set of A0 in T_1 (A0_d[1]) and the delay set of A2 in T_1 (A2_p[1]). This ensures that, in the replay, W1 is not executed until R6 is executed. To infer the destination of a new dependence, the memory count of the destination is sent together with the coherence messages.

Note that the order that the two RAW dependences occur cannot be changed. W4 can only be the source of a dependence when it is globally performed, which is guaranteed to happen after it retires from processor. Instructions retire from processor in order, therefore, before W4 is globally performed, R3 must have already been retired. Therefore, the first RAW from T_1 to T_2 happens before the second RAW from T_2 to T_3.

In RC machine, where the writes can be reordered, we consider how it can affect our mechanism. Let's assume that all the reads in Figure 7 (d) are changed to writes (change R3, R5 to W3, W5). Then all the RAW are changed to WAW. In this scenario, it is possible that the WAW from T_2 to T_3 happens first. Later, when the WAW from T_1 to T_2 is formed as W3 is performed, according to the mechanism in Section 3, we will find W2's address B is contained in the write address set of A1's previous spectrum. Since we know that the memory count of W3 is smaller than the memory count in A1 for T_2, we can infer that a new arch (A0) needs to be created but it should be *before* A1. From the example, we see that, due to the out-of-order memory operation, the new arch can be created before the existing ones. After creating A0 before A1, the algorithm detects the SC violation in the same way.

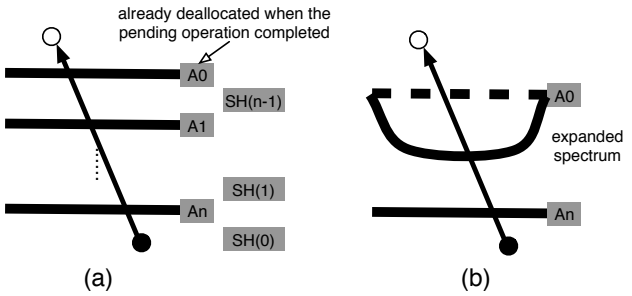

Figure 8: SCV detection based on spectrum

4.3.3. Spectrum History Requirement As been shown in the previous section, Spectrum History (SH) is a key component to detect SC violations. However, the system can only keep a limited number of SH entries. It is possible that at one point, the oldest SH entry has to be deallocated since a new arch (new spectrum) is created. If there is still a pending store operation before that oldest, to-be-deallocated SH, the pending operation may later become the destination of a new dependence when the oldest SH entry is already deallocated. In the normal case, when we still had the entry in SH, the pending store operation may be put into the delay set of the arch of the old SH entry or the value (in case the instruction is a load) is recorded. However, if we have already lost the information of that old arch, there is no way to update the delay set since the SC violation cannot be detected. The problem can be seen in Figure 8 (a).

The essential problem is that each processor has the current oldest pending memory operation (typically a store), the oldest SH entry can only be deallocated when all the memory operations in the spectrum have completed.

We use the following simple way to avoid the early deallocation of old SH entries. The record module keeps the current oldest pending memory operation's count from each processor. When the oldest SH entry is about to be deallocated, the record module consults the oldest pending information. If the oldest one's count is still smaller than the arch's count for that thread, the SH entry cannot be deallocated until all the pending operations before the arch are completed. This may cause the stall of execution when a new dependence needs to create a new arch. With large enough SH entries, the stall happens very rarely. In our experiment, with 16 SH entries for 8-processor system, the stall didn't happen in all the experiments.

4.3.4. Implication of Spectrum Expansion Finally, we show that the spectrum expansion does not affect the SC violation detection scheme. The key insight is that, the expansion will only *increase* the memory count in an existing arch. Therefore, if we can detect a new dependence from the current spectrum to an older spectrum before expansion, we are guaranteed to detect them when more memory operations are moved into the older spectrum. It is because the the old spectrum boundary is pushed down. The insight is shown in Figure 8 (b).

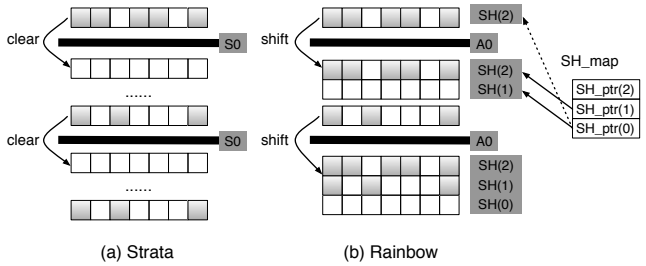

Figure 9: Implementation comparison

5. Implementation

This section discusses the implementation issues. As Rainbow and Strata share the basic assumptions about the handling of page, context switch, checkpoint and cache replacement etc., we will focus on the issues that are specific to Rainbow. Further, we discuss the application of Rainbow techniques to directory-based designs.

5.1. Representing Address Set

To record the write and read set of each spectrum, we can either have extra bits in each cache line or use bloom filters (signatures) to encode the addresses. In the implementation, we use the bloom filters to reduce the hardware cost. Each spectrum has a read and write signature, each of them is 512-bit, we use the signature structure similar to [6].

Figure 9 shows the comparison between Strata and Rainbow. In Strata, the single dependent bit (shown above the strata log) indicates whether the dirty cache line belongs to the previous or current strata region (Figure 9 (a)). The gray entries (dependent bit set) indicate the cache lines are written in the current strata region. When a strata is created, the dependent bit is cleared, therefore, all the future accesses are considered to be in the current strata region after the just created log. In Rainbow, when an arch is created, instead of clearing the current signatures, we copy the write and read signatures to the ones for the previous spectrum, and the signatures for previous spectrum are copied to the older spectrum next to it. In Figure 9 (b), for clarity, we show the case using bits instead of signatures to encode the address sets. On arch creation, the dependent bits for each spectrum are *left-shifted*. In this way, the bits for the current spectrum are cleared and can be used by the new spectrum. Therefore, Rainbow is a generalization of Strata that can track the access information of multiple spectra at the same time. When the new spectrum is created, Rainbow doesn't lose the information of the "previous" current spectrum, while Strata just discards it by flush clearing the dependent bits.

5.2. Spectrum History Management

The manipulation of SH conceptually incurs the move of SH entries when a new arch is created. However, in the implementation, we don't need to copy the records among hardware structures. Instead, we can keep a *SH_map* to correlate the SH records and their spectrum indexes. On the creation of a new arch, the hardware just needs to change the mapping. It is also shown in Figure 9 (b). Initially, SH(2) contains the records for the current spectrum, therefore, SH_ptr(0) is pointed to it. When a new arch (A0) is created, the current spectrum becomes SH(1) and SH(2) is changed to the previous spectrum. To implement the shift of SH entries, the hardware makes SH_ptr(0) points to SH(1) and SH_ptr(1) points to SH(2).

```
req_addr;   /* request physical address */
req_type;   /* RdShd, RdEx, RdCode*/
req_pid;    /* Core id */
req_refs;   /* memory refs */

int tail;   /* spectrum history tail */
int head;   /* spectrum history head */

struct spectrum_history_t {
 int vector_stamp[NCORE];
 addr_set_t wset, rset;
 int delay_set[NCORE][MAX_SET];
 int pending_set[NCORE][MAX_SET];
};

spectrum_history_t SH[24];
/* history depth is 24 */

for (i=tail;i>=0;i--){
 is_conf=conflict(req_addr,req_type,
                  SH[i].wset,SH[i].rset);
 if(is_conf&&i==tail){
  if(tail==NHISTORY-1){
   output_head_to_log();
  }
  create_new_spectrum();
  break;
 }else if(is_conf){
  SH[prev_sh].vector_stamp[req_pid]+=req_refs;
  req_type==WRITE?
  add_to_addr_set(SH[prev_sh].wset, req_addr)
 :add_to_addr_set(SH[prev_sh].rset, req_addr);
  break;
 }
 prev_sh = i;
}
```

Figure 10: Rainbow record algorithm

5.3. Record and Replay Algorithm

In Figure 10 and Figure 11, we show the record and replay algorithms of Rainbow. The operations to modify the delay and pending set and SC violation detection are not included due to the limit of space.

5.4. Applying Rainbow to Directory

Rainbow is based on Strata and they are suitable to be implemented in the snoopy protocol. However, as shown in [20], Strata can be also applied to the directory protocol, which can

```
global barrier b1, b2;
global curr_spectrum;
local refs;
local pid;

while(1){
 barrier(b1);
 if(coreid==0){
  curr_spectrum=pop_spectrum_from_log();
  curr_spectrum or break;
 }

 barrier(b2);
 refs =
   get_refs_from_spectrum(curr_spectrum, pid);
 execute(refs);
 /* executes #refs memory ops */
}
```

Figure 11: Rainbow replay algorithm

use either centralized or distributed directory modules. In this section, we show that the large part of Rainbow techniques can be also applied to the directory protocol.

We partition the techniques in Rainbow into two parts: (1) Expandable spectrum, and (2) support for out-of-order execution and relaxed memory model. For the directory design, we also consider two cases: (1) centralized directory, and (2) distributed directory. We show that, *all* the Rainbow techniques can work smoothly with the centralized directory. For the distributed directory, the expandable spectrum is applicable but the support relaxed memory model is still an open problem.

For the centralized directory, similar to Strata [20], the logs are recorded in the directory module. While the directory cannot observe all the memory operations from every processor, it does observe all the missed accesses since it is served as the serialization point of the accesses to a cache line. The potential dependences can occur on those cache misses. Each memory access count represents the last time the processor communicates with the directory and an arch is logged using these memory counts. Some of the memory counts in the vector can be stale when an arch is logged, it is not a problem because the smaller (not up-to-date) counts are always valid in terms of capturing the shared memory dependences. More details and examples can be found in [20]. Different from Strata, Rainbow has a set of signatures for SHs in the directory module to maintain the address sets. It is easy to see that the algorithm to detect the SC violation can be directly applied with the arches of all SHs. An SC violation is always caused by a new dependence, when it is from a memory operation in the newer spectrum to an older one. Such new dependence can always be observed by the directory.

With distributed directory, each directory module maintains its own signatures and arches for all the SHs. They can perform the spectrum expansion as normal. Without relaxed memory model support, Rainbow works correctly in distributed directory because it still ensures the following property: each cross-node dependency is separated by at least one arch. The spectrum expansion is a method to preserve the above property with *less* logs. The distributed directory requires an offline analysis to eliminate the overlapped spectra and generate the eventual merged logs for replay. The offline analysis in Strata is directly applicable to Rainbow's arches, since they ensure the same property.

Finally, we point out that the current Rainbow techniques to support relaxed memory model (and out-of-order execution) are *not* applicable to the distributed directory design. The challenge is due to the overlapped spectra. The problem is that, both the distributed directory (due to outdated memory access count from the processors) and the SC violations can cause the overlapped spectra. The current Rainbow record phase doesn't record enough information for the offline analysis to distinguish them. A potential solution is to update the memory count in every directory modules on every cache miss. This incurs substantial extra messages but avoids the overlapped spectra due to distributed directory (therefore, all such regions are due to SC violation). We leave it to our future work.

6. Evaluation

6.1. Evaluation Setup

We use Simics to model an x86-based chip multiprocessor with private L1 cache that are kept coherent with a shared L2 cache using a snoopy-based MESI cache coherence protocol. Table 1 shows the parameters of the architecture. We perform the parallel record and replay runs with our applications running on 8 processors. We use at most 24-entry spectrum history (SH) and 512-bit bloom filters for the read and write sets of each SH. Another 512-bit bloom filter is used to record the replaced lines. We execute 10 applications from the SPLASH-2 suite.We also implemented Strata for comparison.

Processor and Memory System Parameters	
Chip multiprocessor	Bus-based with snoopy MESI protocol Rainbow recorder is connected to the bus
Processor	Four issue, x86 ISA
L1 Cache	64KB size, 64B line, 2-way assoc. 1 cycle hit, 2-bit LRU replacement
L1 Cache Miss Latency	10-cycle round-trip to another L1 10-cycle round-trip to L2
L2 Cache	Shared, 2M size, 64B line, 8-way assoc. 5 cycle hit
L2 Cache Miss Latency	100-cycle round-trip to memory
Bus width	8B
RainbowParameters	
Read & write signature	Two 512-bit Bloom filters per SH
Eviction signature	512bits Bloom filter
# Spectrum History	4,8,16,24

Table 1: Parameters of the simulated hardware.

6.2. Log Size Reduction

Figure 12 shows the log size reduction by Rainbow compared to Strata with different SH sizes. On average, Rainbow reduces the log size by 17.6% (4 SH), 19.7% (8 SH), 24.1% (16 SH)

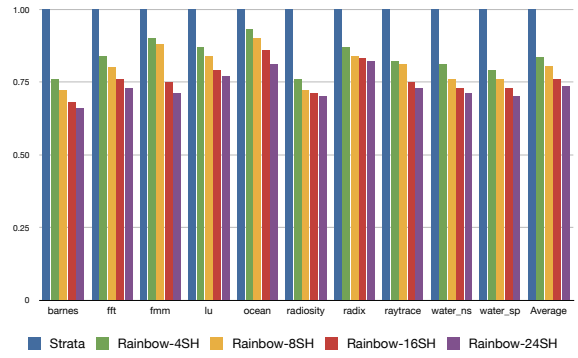

Figure 12: Rainbow log size reduction

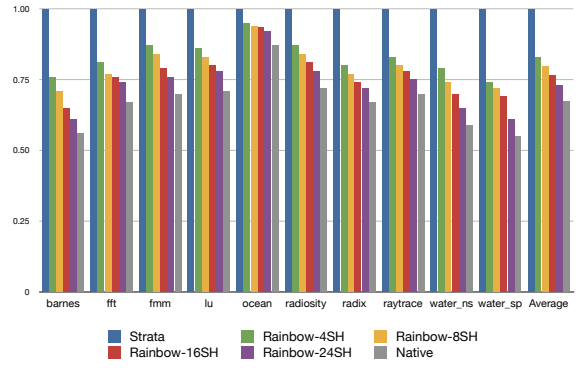

Figure 13: Rainbow replay speed

and 26.6% (24 SH). The reduction is due to more frequent reuse of the existing arches to capture the new dependences.

We also see that as the number of SH entries increases, the log reduction gain is diminishing. It is because, due to the inter-thread dependences or per-thread program order, most of the memory operations cannot move across a large number of spectra. From the results, we can see that the SH sizes of 4 to 8 appear to be the best choices.

Overall, we report that Rainbow's log compact technique is effective in reducing the log size with small hardware cost.

6.3. Replay Speed

We compare the replay speed with Strata and the native execution. The result is shown in Figure 13. For each application, the execution time is normalized to Strata. On average, Rainbow increases the replay speed over Strata by 17.2% (4 SH), 20.4% (8 SH), 23.4% (16 SH) and 26.8% (24 SH) and incurs a slow down of 8.6% compared to the native execution. The replay speed increase follows the same trend as the log size reduction, since smaller number of logs results in larger spectrum and enables more parallelism.

For one application, ocean, the replay speed increase is not very significant. It is mainly due to the global synchronization operation in this application. In particular, some memory operations can be moved up across several spectra but in the replay, the reduction in the number of logs doesn't translate to much higher replay speed.

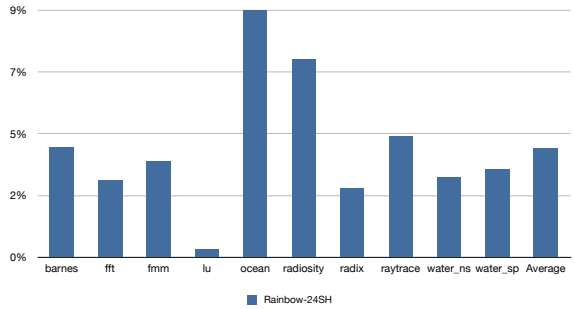

Figure 14: Rainbow extra log size for out-of-order information

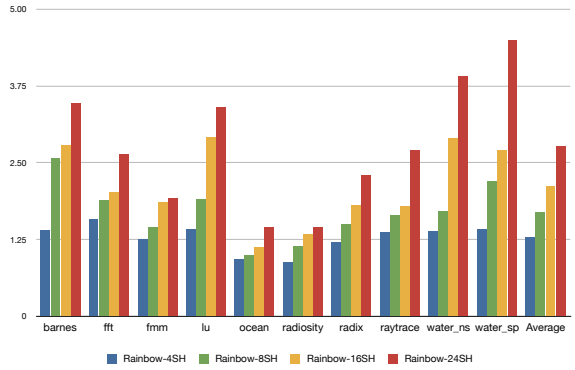

Figure 15: Rainbow memory operation move length

6.4. Extra Log for Out-Of-Order Execution

Figure 14 shows the extra log used to record the information for the out-of-order execution effects. Such log includes the load values in the overlapped and incompatible spectra and the delay/pending set. We show the percentage of the extra log with the full size of logs (arches + extra log for out-of-order memory operations). The results are based on 24-entry SH. We found that the extra log size is nearly irrelevant to the number of SH entries.

We see that for most of applications, the extra log is under 5%. This implies that the out-of-order effects are not significant. It is intuitive, since in the small-scale snoopy protocol, we don't expect to see drastic memory access reordering due to the small variance in memory access latency. For ocean and radiosity, the log size is larger. The elevated level of reordering is mainly due to the memory operations in synchronization functions.

Although the results show that the reordering effects are not significant, we believe it is critical to support relaxed memory model and record such events. They tend to be the the most critical information needed by the programmers in debugging.

6.5. Spectrum Expansion Characterization

Figure 15 shows the average number of spectra that a new memory operation can move across for each application. We see that larger numbers of SHs increase the chance to move further. However, the longer length of move may not translate to smaller log size or high replay speed. It is because fewer

moves may achieve similar effect of log elimination. In several cases, although some memory operations can be moved across several spectra, a new memory operation is still stopped at the same position due to conflicts.

6.6. Overhead Sensitivity

Appl.	Total Ins (M)	Mem Ops (%)	Total Bus Acc (M)	OOO Log Bus (%)	Exec. Time Overhead (%) (SH=4)
barnes	2196.6	39.3	33.0	1.4	0.1
fft	6.2	23.7	1.3	0.9	0.3
fmm	3709.3	34.3	26.3	1.1	0
lu	232.4	16.4	1.8	0.07	0.4
ocean	134.6	23.7	14.7	1.7	0.8
radiosity	529.5	36.9	27.3	1.5	0.6
radix	45.4	14.2	3.6	1.3	0
raytrace	245.6	32.5	5.8	0.9	0.4
water_ns	145.3	24.4	10.4	0.8	0
water_sp	125.4	28.4	8.5	0.7	0

Table 2: Rainbow's execution overhead for 8 cores.

Finally, we show the execution characteristics and overheads in Table 2. The results are for the SH size of 4. The overhead is the execution stall due to the wait for the old pending memory operations before the oldest spectrum to close. For each application, we also show the total number of instructions, the percentage of memory operations, total bus accesses and the bus accesses due to the log of out-of-order information.

We see that the overhead in logging the out-of-order execution information is very small, in all applications, it is under 2%. Even for very small SH sizes, the overhead due to the wait for old pending operations is extremely small. In several applications (e.g. fmm, radix, water_ns, water_sp), we didn't encounter the stall at all.

7. Conclusion

This paper proposes Rainbow, the first R&R scheme that supports any memory model (not only TSO) suitable for snoopy protocol. It is based on Strata but improves in two key aspects. First, it records near-precise happens-before relations, reducing the number of logs and increasing the replay parallelism. Second, it supports record and replay with any relaxed memory consistency model. The goals are achieved by two key techniques. First, expandable spectrum allows younger non-conflict memory operations to be moved into older spectrum, so that fewer arches are created and logged. Second, we propose the novel SC violation detection scheme based on the existing arches to detect the overlapped and incompatible spectra. Extra logs are created to record the out-of-order execution effects. Most of the Rainbow techniques are also applicable to the directory-based protocol. Using simulations, we show that, on average, Rainbow reduces the log size compared to Strata by 26.6% and increases the replay speed by 26.8%. The SC violations are few but do exist in the applications evaluated.

References

[1] H. Agrawal *et al.*, "An Execution-Backtracking Approach to Debugging?" *IEEE Software*, vol. 8, no. 3, May 1991.

[2] G. Altekar and I. Stoica, "ODR: Output-deterministic Replay for Multicore Debugging," in *Symp. on Operating Systems Principles*, 2009.

[3] D. F. Bacon *et al.*, "Hardware-Assisted Replay of Multiprocessor Programs," in *ACM/ONR Workshop On Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 1991.

[4] A. Basu *et al.*, "Karma: Scalable Deterministic Record-Replay," in *International Conference on Supercomputing*, 2011.

[5] T. Bressoud and F. Schneider, "Hypevisor-based Fault-tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, Feb 1996.

[6] L. Ceze *et al.*, "BulkSC: Bulk Enforcement of Sequential Consistency," in *Int. Symp. on Computer Architecture*, 2007.

[7] Y. Chen *et al.*, "LReplay: A Pending Period based Deterministic Replay Scheme," in *Int. Symp. on Computer Architecture*, 2010.

[8] J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," in *Symposium on Parallel and Distributed Tools*, 1998.

[9] B. Cully *et al.*, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[10] G. W. Dunlap *et al.*, "ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay," in *Symposium on Operating Systems Design and Implementation*, 2002.

[11] ——, "Execution Replay of Multiprocessor Virtual Machines," in *International Conference on Virtual Execution Environments*, 2008.

[12] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models." August 1991, pp. 355–364.

[13] D. R. Hower and M. D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," in *Int. Symp. on Computer Architecture*, 2008.

[14] S. T. King and P. M. Chen, "Backtracking intrusions," in *Symposium on Operating Systems Principles*, 2003.

[15] S. T. King *et al.*, "Debugging operating systems with time-traveling virtual machines," in *USENIX Annual Technical Conference*, 2005.

[16] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Comput.*, vol. 36, no. 4, Apr. 1987.

[17] D. Lee *et al.*, "Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism," in *Architectural Support for Programming Languages and Operating Systems*, 2010.

[18] P. Montesinos *et al.*, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *Int. Symp. on Computer Architecture*, 2008.

[19] ——, "Capo: a Software-Hardware Interface for Practical Deterministic Multiprocessor Replay," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[20] S. Narayanasamy *et al.*, "Recording Shared Memory Dependencies Using Strata," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[21] S. Park *et al.*, "PRES: Probabilistic Replay with Execution Sketching on Multiprocessors," in *Symp. on Operating Systems Principles*, 2009.

[22] G. Pokam *et al.*, "Architecting a Chunk-based Memory Race Recorder in Modern CMPs," in *Int. Symposium on Microarchitecture*, 2009.

[23] ——, "CoreRacer: A Practical Memory Race Recorder for Multicore X86 TSO Processors," in *Int. Symp. on Microarchitecture*, 2011.

[24] M. Russinovich and B. Cogswell, "Replay for Concurrent Non-deterministic Shared-memory Applications," in *Programming Language Design and Implementation*, 1996.

[25] S. M. Srinivasan *et al.*, "Flashback: a Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *USENIX Annual Technical Conference*, 2004.

[26] K. Veeraraghavan *et al.*, "DoublePlay: Parallelizing Sequential Logging and Replay," in *Architectural Support for Programming Languages and Operating Systems*, 2011.

[27] G. Voskuilen *et al.*, "Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording," in *Int. Symp. on Computer Architecture*, 2010.

[28] M. Xu *et al.*, "A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay," in *Int. Symp. on Computer Architecture*, 2003.

[29] ——, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.