Pacifier: Record and Replay for Relaxed-Consistency Multiprocessors with Distributed Directory Protocol *

Xuehai Qian⋆ Benjamin Sahelices† Depei Qian‡
⋆ University of California Berkeley
† Universidad de Valladolid ‡ Beihang University
xuehaiq@berkeley.edu benja@infor.uva.es depeiq@buaa.edu.cn

Abstract

Record and Deterministic Replay (R&R) of multithreaded programs on relaxed-consistency multiprocessors with distributed directory protocol has been a long-standing open problem. The independently developed RelaxReplay [8] solves the problem by assuming write atomicity.

This paper proposes Pacifier, the first R&R scheme to provide a solution without assuming write atomicity. R&R for relaxed-consistency multiprocessors needs to detect, record and replay Sequential Consistency Violations (SCV). Pacifier has two key components: (i) Relog, a general memory reordering logging and replay mechanism that can reproduce SCVs in relaxed memory models, and (ii) Granule, an SCV detection scheme in the record phase with good precision, that indicates whether to record with Relog. We show that Pacifier is a sweet spot in the design space with a reasonable trade-off between hardware and log overhead. An evaluation with simulations of 16, 32 and 64 processors with Release Consistency (RC) running SPLASH-2 applications indicates that Pacifier incurs 3.9% $\sim$ 16% larger logs. The slowdown of Pacifier during replay is 10.1% $\sim$ 30.5% compared to native execution.

1. Introduction

Record and Deterministic Replay (R&R) of multithreaded programs is a technique that involves logging sufficient information in a parallel execution and replaying it later deterministically. R&R has many uses in program debugging, intrusion analysis and fault-tolerant, highly-available systems. R&R requires logging two sources of non-deterministic events during execution: external inputs (e.g., results of system calls) and the interleaving of shared memory accesses from different processors, which typically needs hardware assistance due to high software overhead. This hardware component is called Memory Race Recording (MRR).

Early MRRs record memory races when processors *do* communicate [23, 24], so they require substantial hardware to restrict log size. Recent chunk-based approaches alternatively record periods that processors *do not* communicate. The sequence of communication-free dynamic instructions from a processor is called an *episode* [3, 9], *chapter* [22] or *chunk* [7, 12, 15, 16]. Another approach is based on the concept of global log (i.e., Strata [14] or Arch [17]) that creates time layers across all the running threads. A single log can potentially capture many dependences.

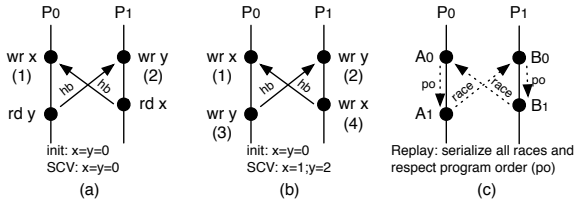

Figure 1: Examples of SCV and the Implications to R&R

Despite many years of research, R&R still has a long-standing open problem: **R&R** *on* **R***elaxed* **M***emory* **M***odel with* **D***istributed* **D***irectory-based protocol*. In this paper, we call it $R^3M^2D^2$. We assume a multiprocessor system with cache coherence but no write atomicity, which represents the properties of relaxed-consistency models in current processors (e.g., PowerPC [20] and ARM [2]).

Memory consistency models specify the order in which memory accesses performed by one processor become visible to other processors. Sequential Consistency (SC) [11] is a strong and intuitive memory model that requires the memory accesses of a program appear to have executed in a global sequential order consistent with the per-processor program order. Since SC limits many hardware and compilers optimizations, most modern multiprocessors support a relaxed-consistency model, where the memory access interleaving may not confirm to SC. We call these behaviors SC Violations (SCVs). Figure 1 (a) shows an example. The arrows indicate the happens-before (*hb*) relation between two conflict accesses. This interleaving, which is allowed by Total-Store-Order (TSO) [1], makes the results of the two loads in P_0 and P_1 still 0. It is unintuitive and not allowed in SC. Figure 1 (b) shows an SCV in Release Consistency (RC) [1]. It is possible that x and y are eventually 1 and 2, respectively, because RC allows out-of-order stores.

Given that one of the most popular uses of R&R is to debug parallel programs and that the relaxed-consistency model is an important cause of concurrency bugs [13], R&R should completely support such models.

The essential implication of relaxed memory model on R&R is that sequential replay cannot reproduce SCV. Shown in Figure 1 (c), the SCVs are the execution behaviors that do not conform to sequential order. They are manifested as *cycles* with edges due to conflict accesses (*race*) and program orders (*po*). A typical replayer respects program order and can only

*This work was supported in part by the National Science Foundation of China under grants 61073011 and 61133004, Spanish Gov. European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050.

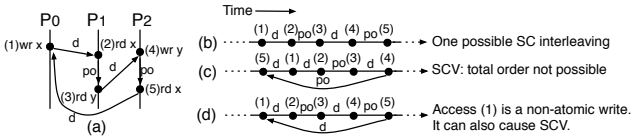

Figure 2: Non-SC execution and non-atomic writes

sequentially reproduce data races.[1] Consequently, when the replayer tries to reproduce a data race (e.g., $A_1 \rightarrow B_0$), some previous instructions (e.g., A_0) may have been executed, but the original execution requires that A_0 is executed after B_1.

Among existing schemes, RTR [24], Rerun [9], CoreRacer [16] and Karma [3] can handle TSO, so they are able to record and replay the case in Figure 1 (a). Rainbow [17] is the first scheme supporting memory models beyond TSO: it can handle RC in Figure 1 (b) but does not work with distributed directory. RelaxReplay [8] can handle RC and work with distributed directory. However, RelaxReplay assumes write atomicity, which is not supported in popular commercial processors such as PowerPC [20] and ARM [2]. Therefore, *there is still no solution to $R^3M^2D^2$ without write atomicity*.

This paper proposes *Pacifier*, the first complete chunk-based solution to $R^3M^2D^2$ without assuming write atomicity. It has two key components, Relog and Granule. Relog is a general mechanism to log and replay reorderings in a chunk-based R&R scheme. The key idea is to delay a subset of instructions in a chunk until a remote chunk completes. Relog essentially enforces the reordering of the delayed instructions and the other instructions in the chunk. The ordering between a local instruction with a remote chunk is the key to supporting non-atomic writes. Granule tries to minimize logged reorderings while still reproducing all SCVs. It distinguishes the reorderings that do not cause SCVs conservatively and avoids generating logs in those cases. Granule is based on the chunk termination policy in Karma [3] that naturally filters out the dependences in a single direction. Granule represents a reasonable trade-off between hardware and log overhead.

We evaluate Pacifier with simulations of 16, 32 and 64 processors with RC running 10 SPLASH-2 applications. Pacifier is compared against Karma (with no SCV detection) and a hypothetical configuration using Volition [18] as the precise SCV detector. The results show that Pacifier incurs 3.9% $\sim$ 16% larger logs compared to Karma due to SCVs. The slowdown of Pacifier during replay is 10.1% $\sim$ 30.5% compared to native execution. The small log size is due to Granule's ability to distinguish likely SCVs from local reorderings invisible to remote processors. The results indicate that Pacifier is an efficient solution to $R^3M^2D^2$.

This paper is organized as follows: Section 2 provides the problem statement and the existing R&R for relaxed memory models. Section 3 presents the ideas of Relog and Granule. Section 4 shows the implementation of Pacifier, including the hardware structures and record and replay algorithm. Section 5 compares Pacifier with other schemes. Section 6 evaluates Pacifier. Section 7 concludes the paper.

[1] Some schemes (e.g., Karma [3], Strata [14] and Rainbow [17]) allow parallel replay, but that is for race-free regions.

2. Background and State-of-the-art

2.1. Problem Statement

In a multiprocessor execution, memory accesses from different processors form a dependence graph. An edge in the graph represents either local program order ($\xrightarrow{po}$) or inter-processor data dependence ($\xrightarrow{d}$). The record phase dynamically constructs and logs the graph, and the replay phase uses the logs to reproduce the same execution. The program order can be trivially enforced with no record. The efforts in designing R&R schemes have been focusing on recording data races. It is further complicated by relaxed-consistency and non-atomic writes. The following concepts are defined to provide a context of discussion.

Time. We assume a global physical clock to specify the order of accesses. It is only used in the explanation and not required in the multiprocessor system.

Performing a Memory Request by a processor P_i (defined in [6]).

- A **Load** by P_i is considered *performed with respect to (w.r.t.) P_k* at a point in time when the issuing of a Store to the same address by P_k cannot affect the value returned by the Load.
- A **Store** by P_i is considered *performed w.r.t. P_k* at a point in time when an issued Load to the same address by P_k returns the value defined by this Store (or a subsequent Store to the same location).
- An access is *performed* when it is performed w.r.t. all processors. We sometimes use *globally performed* to avoid confusion, but they are interchangeable. We denote the performed time of a memory request "i" as T_p(i).

Sequential Consistency (SC). There is always a total order among all accesses [11]. Figure 2 (b) shows one of the valid SC interleavings of the accesses from three processors in Figure 2 (a).

Relaxed-Consistency. The total order may not exist due to SCVs. An SCV is a cycle among some accesses in the dependence graph. Figure 2 (c) shows an example.

Non-atomic Writes. Write atomicity means that a write operation by a processor P_i is performed w.r.t. another processor P_j only if it has been performed w.r.t. all processors, and that writes to the same location are serialized [21]. Cache coherence does not imply write atomicity, because it only guarantees the serialization of accesses to the same location but says nothing about when the values are visible. We observe the following implication of non-atomic write on dependence graph construction:

During dependence graph construction, write atomicity ensures that *after a node for a write in the graph becomes the source of a $\xrightarrow{d}$, it cannot be the destination of any other $\xrightarrow{d}$*. For a non-atomic write, *the write node can first become the*

source of a $\xrightarrow{d}$ *and then the destination of another* $\xrightarrow{d}$.

Figure 2 (d) shows an SCV due to a non-atomic write. In this case, (1) is first performed w.r.t. P_1 and (2) reads the value produced by (1). At this point in time, (1) has *not* been performed w.r.t. P_2. Later, (5) in P_2 reads the old value of address "x". Eventually, (1) is performed w.r.t. P_2 and since (5) already read the old value, a Write-After-Read (WAR) dependence is formed from (5) to (1). This dependence closes a cycle in the dependence graph.

Note that this cycle involves the same set of accesses as the case in Figure 2 (c), but if the system ensures write atomicity, the SCV cannot happen. The key factor is the order of accesses. With write atomicity, if (1) $\xrightarrow{d}$ (2), it is guaranteed that (1) is also performed w.r.t. P_2 and (5) should read the new value. Therefore, (5) $\xrightarrow{d}$ (1) cannot happen. However, in the case of non-atomic writes, as shown in Figure 2 (c), (5) can read the old value of *x* before (2) reads the value produced by (1). While the SCV looks similar, the cause is different.

In summary, non-atomic writes can cause additional SCVs that do not exist assuming write atomicity. Next, we explain how existing R&R schemes record data dependences, with an emphasis on the handling of relaxed-consistency.

2.2. R&R SC Execution

Early point-to-point approaches [23, 24] record memory races when threads *do* communicate, incurring large logs and overhead. To reduce overhead, chunk-based techniques [3, 5, 7, 9, 12, 15, 16, 22] were proposed. They record the periods that threads *do not* communicate. This approach normally incurs negligible recording overhead and can achieve very small logs. In this paper, we uniformly call a sequence of communication-free dynamic instructions a *chunk*. The state-of-the-art of this approach is Karma [3], which uses Directed Acyclic Graph (DAG) to achieve parallel replay and smaller log size at the same time. Timetraveller [22] achieves similar goals with post-dating. Delorean [12] enables fast recording and parallel replay, but it is based on a non-conventional cache coherence protocol that has not yet been implemented.

Strata [14] shows another approach of R&R. It requires that all the processors agree to start a new strata region when there is a dependence between any two processors. Strata uses a recording approach that requires that all processors record an entry in their logs at the same time. Rainbow [17] optimizes Strata by recording near-precise happens-before relations, reducing the size of logs and increasing replay parallelism.

2.3. R&R Relaxed-Consistency Execution

To reproduce a dependence graph, the replay phase enforces one edge (either $\xrightarrow{d}$ or $\xrightarrow{po}$) at a time. Obviously, with a cycle (i.e., SCV), such replay is not possible because an access required to happen after another has already been executed.

2.3.1. Related Concepts We define a set of concepts used in understanding existing schemes and the discussion of Pacifier in Section 3.

Sequence Number (SN). Each processor can assign a monotonically increasing SN to every memory access according to the program order. The SN of an instruction "i" is denoted as SN(i). SN may not exist in the current processors or other R&R schemes, but it is needed in Pacifier. The implementation of SN is trivial.

Reordered Instructions. If $SN(i_0) < SN(i_1)$ and $T_p(i_0) > T_p(i_1)$, i_0 and i_1 are *reordered*.

Pending Window (PW). In each processor, PW includes a set of instructions that are either not performed or there exist previous instructions in $\xrightarrow{po}$ not performed. Intuitively, PW is a superset of the instructions in Reorder Buffer (ROB), because after a store is retired from the processor, it may stay in the store buffer while being globally performed. This will make the store itself and the following retired instructions in PW. Instructions always enter and are removed from PW *in program order*. PW (or similar structures) does not exist in current processors but is used in Rainbow [17] and RelaxReplay [8]. They are essentially the same as Active Table (ACT) in Volition [18].

Completion. A memory operation is completed when it is removed from the PW. The completion time point for an instruction "i" is denoted as T_c(i).

Delay Set (D_{set}) and Pending Set (P_{set}). D_{set} indicates the instructions that need to be "skipped" in a region of consecutive dynamic instructions in a processor. P_{set} indicates that the instructions earlier in $\xrightarrow{po}$ need to be executed before a new region starts.

2.3.2. How to R&R the cycles? For a R&R scheme to handle SCV cycles in a dependence graph, all the current solutions *"relax"* $\xrightarrow{po}$ to reproduce the cycles. This is exactly how the original execution produces such cycles. A relaxed-consistency processor can reorder the instructions in $\xrightarrow{po}$ for higher performance (e.g., to hide write latency). If no remote conflict access is interleaved with reordered instructions in PW, such reordering is not visible by any remote processors. Otherwise, local reordering is observed by remote processors, which may or may not cause SCV cycles.

Therefore, a R&R scheme supporting relaxed-consistency model needs to have three extra components:

- *SCV Detection (SCV-D)* detects cycles (SCVs) in recording.
- *SCV Logging (SCV-L)* records reordered instructions required to reproduce the SCVs during replay.
- *SCV Replay (SCV-R)* faithfully reproduces the SCVs using the logs generated by SCV-L.

We compare the existing R&R schemes in Table 1 based on the above three aspects. Depending on the relaxed-consistency model supported, different R&R schemes may enforce different types of local reordering of $\xrightarrow{po}$.

Total-Store-Order (TSO). This model only allows the reordering of load and earlier stores in $\xrightarrow{po}$ [1] (i.e., allowing a load to be performed while earlier stores are still being per-

R&R schemes	SCV Detection (SCV-D)	SCV Logging (SCV-L)	SCV Replay (SCV-R)	Limitations
RTR [24] (also applicable to Karma [3] and Rerun [9])	Monitor the incoming invalidations to the addresses of the speculatively retired loads. A potential SCV is found when any such load address is invalidated.	Log the load value of the potential SC-violating loads.	Use the logged value to overrule the value loaded from memory during replay.	Only support TSO. Reason: only considers the store-to-load reordering.
CoreRacer [16]	When a chunk is logged, the store buffer is not empty.	Log the number of pending stores in store buffer in a counter, RSW.	Execute (cs+RSW) instructions, leave (RSW) stores in the simulated store buffer.	Only support TSO. Reason: RSW can only capture the number, but not the order, of pending stores.
Rainbow [17]	For a new dependence, when the region of the destination access is younger than the region of the source access.	D_{set} and P_{set}. Loads in D_{set} are logged as load values, since loads cannot be skipped.	For a region, first execute the instructions in P_{set} and skip the instructions in D_{set}.	Only work for snoopy or centralized directory protocol: SCV-D is based on a centralized algorithm.
RelaxReplay [8]	Base: when PISN $\neq$ CISN. Opt: only when there is a conflict remote access performed w.r.t. the local processor between the perform and count time of an access. *Comments:* PISN: Performance Interval Sequence Number. It is the chunk when an access is performed. CISN: Current Interval Sequence Number. An instruction is counted with CISN at T_c.	(1) Mark the reordered access as a dummy entry in the chunk; (2) Record the distance between CISN and PISN as offset; (3) For a reordered load, record the load value; (4) For a reordered store, record the address and store value;	For reordered loads, read the values from logs. For reordered stores, need an offline "patching" step to insert them into the interval when it is performed. In the replay, execute them together with the earlier interval.	(1) Using data races as the proxies for SCV, this has been shown to have false positives [13, 18]; (2) Potentially large log size due to the SCV-D false positives; (3) Cannot handle non-atomic writes — assumes only one performed point.

Table 1: Existing Support for Relaxed Memory Models in R&R Schemes.

formed). R&R schemes supporting TSO only need to log and enforce this single type of reordering.

RTR [24] is the first R&R scheme that supports TSO. The idea is to detect SCVs conservatively and remember the values of the potentially SC-violating loads. This approach can also be applied to Rerun [9] and Karma [3]. Referring to Figure 2 (a) and (c), the cycle contains the reordered execution of (4) and (5). This can be recorded by logging the load value of (5) and replayed by letting the recorded value overrule the value loaded from memory.

CoreRacer [16] proposes an interesting alternative technique to record the reordered instructions. Instead of logging the load values, it uses the simulated store buffer to delay the pending stores. In the record phase, when a processor needs to terminate a chunk (with the boundary decided by the retired instructions, not completed ones) and log the instruction count, it checks whether the store buffer is empty, and if it is not, it records the number of pending stores in the store buffer in a counter, RSW. The format of a chunk is <CS, RSW, TS, type>, where CS is the chunk size, TS is the timestamp, and type distinguishes whether the chunk is an SC or TSO-chunk (which can be implied by RSW). In the replay phase, RSW and type are used to delay the pending stores for the TSO-chunks.

Release Consistency (RC). RC is more relaxed than TSO since it allows all kinds of instruction reordering [1], in particular, the reordering of stores. RC can be handled by two existing schemes, one of them developed concurrently with this paper.

Rainbow [17] is the first R&R scheme supporting RC. It uses a history of the generalized form of Strata logs [14] (i.e., arches) to detect SCVs. This technique is also used to reduce log size and increase replay speed. Rainbow can detect SCVs from RC. The reordering is recorded by D_{set} and P_{set}. This technique is general enough to record any kinds of reordering. The main issue is that it depends on centralized mechanisms and can only work well for snoopy bus or centralized directory.

RelaxReplay [8] was developed concurrently with this paper. It is the first R&R scheme supporting RC with directory assuming write atomicity. It introduces the concept of an *Interval*, which is similar to a chunk but the boundary is determined by the inter-processor communication. The interval when an instruction is *performed* and *counted* (in program order when an instruction completes) is identified by PISN and CISN, respectively. When RelaxReplay approximately detects an SCV, the reordering information is recorded. The Base SCV-D scheme reports an SCV if (CISN $\neq$ PISN) when an instruction is counted. The Opt SCV-D scheme reports an SCV only when there is a conflict remote access performed w.r.t. the local processor between the perform and count time.

3. Pacifier: The Insights

3.1. Understanding Non-atomic Writes

We assume a multiprocessor system with directory-based cache coherence supporting RC. This memory model serves as an example of relaxed-consistency models since it allows all kinds of reordering, but the techniques can be used for other models. We do not assume write atomicity. The non-atomic writes present a unique challenge to R&R. Next, we take a closer look at the behavior of non-atomic writes.

We first show the behavior of atomic writes in a directory protocol in Figure 3 (a). P_0 first starts a write to A ($wr\ A$). It involves sending invalidations to the sharers and receiving acknowledgments. When a sharer invalidates the cache line

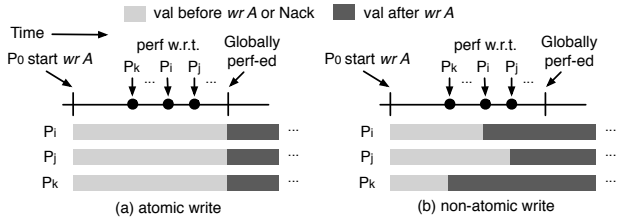

Figure 3: Understanding Non-atomic Writes.

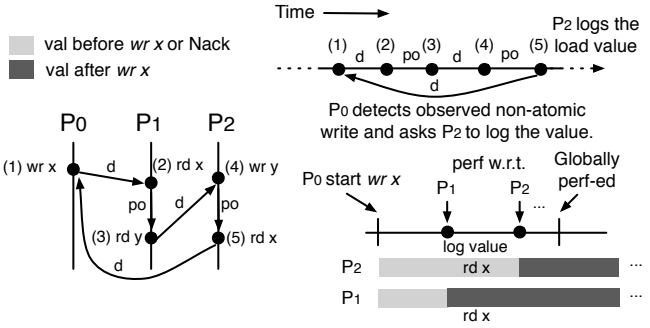

Figure 4: Logging the Effects of Non-atomic Writes.

containing A, *wr A* is performed w.r.t. the processor of the sharer. We consider three processors P_i, P_j and P_k among all the sharers. *wr A* is performed w.r.t. them at different points in time and at a later point, *wr A* is eventually globally performed. With write atomicity, even if *wr A* has already been performed w.r.t. a processor, that processor cannot observe the new value of A until the write is globally performed. Before that, the load from that processor will either get the old value (i.e., a load hit in cache) or be Nack-ed (i.e., wait until the global perform point).

The case for non-atomic writes is shown in Figure 3 (b). The only difference is that when *wr A* is performed w.r.t. a processor, that remote processor can read the new value of A immediately. Therefore, there exists a time period in which one processor observes the new value while another processor still observes the old value. It is important to note the following property. While a remote processor (e.g., P_k) can *read* the new value of A when the others (e.g., P_i) still observe the old value, P_k *cannot start a new write* to the cache line of A until *wr A* from P_0 is globally performed. This property is ensured by cache coherence.

3.2. Logging the Effects of Non-atomic Writes

We assume that R&R can be performed on machines with different memory models. In particular, even an SC machine with write atomicity should be able to replay an execution in a RC machine without write atomicity. Therefore, there is no way to replay the different performed points in Figure 3 (b). We propose the following solution.

The effects of a non-atomic writes only need to be recorded if they are "observed" by remote processors. Consider the example in Figure 4, which is the same as the one in Figure 2 (a). The non-atomicity of *wr x* is observed when the new and old values are observed by P_1 and P_2, respectively, before the write is globally performed. When P_0 receives the invalidation acknowledgement from P_2 (*wr x* is performed w.r.t. to P_2), if P_0 remembers that the new value was read by P_1, P_0 can conclude that the effects of the non-atomic write were observed. Since there is no way to simulate non-atomicity, P_0 sends an additional message to P_2 requesting the processor *log the (old) value that it previously read*. In addition, this WAR will not incur a new order from the chunk in P_2 to P_0.

The idea is illustrated in Figure 4. (2) in P_1 and (5) in P_2 are the two reads that observe the new and old values of x during the time period that the non-atomic write is performed. P_2, which observed the old value, needs to log the load value.

This technique will not incur much log size increase because it is only needed in a very special pattern — when the effects of non-atomic writes are visible. Also, it will not incur many extra messages because the extra message exchange is only needed if a performed load in PW is invalidated before it is removed from PW. Indeed, there are some "unnecessary" message exchanges because in most cases, the processor (e.g., P_2) will find no need to record the value.

To ensure that the (old) value is still available when a processor is asked to log it, the processor which has performed loads in PW with the value later invalidated (e.g., P_2) needs to wait for the extra response from the writer (e.g., P_0) before it can remove the entry from PW. Note that the additional requirement will only slightly increase the size of PW and the processor does not need to stall.

3.3. Relog: R&R the Reordered Execution with Chunk

3.3.1. Assumptions for Chunks In Pacifier, a *chunk* means a sequence of communication-free dynamic instructions from a processor. We describe the notations and properties of a general chunk-based R&R scheme.

For P_i, the sequence of chunks is denoted as C_i[i], C_i[i+1],... We use C_i to denote any chunk in P_i. The chunk in P_i that contains instruction I_k is denoted as $C_i(I_k)$. We can omit the subscript i and denote it as $C(I_k)$ when the context is clear. In addition to $\xrightarrow{d}$ and $\xrightarrow{po}$ as we defined before, we use $\xrightarrow{r}$ to denote replay order. This can be used between two chunks, two instructions, or an instruction and a chunk. If C_i is required to execute before C_j in the replay, we have $C_i \xrightarrow{r} C_j$. If $I_0 \xrightarrow{po} I_1$ needs to be reordered, then $I_1 \xrightarrow{r} I_0$. If an instruction I_0 needs to execute after a remote chunk C_{rmt}, then $C_{rmt} \xrightarrow{r} I_0$. The start and end of a chunk (C_i) are indicated by the SN of the first and last instruction of C_i: $SN_s(C_i)$ and $SN_e(C_i)$.

Dependence Property. I_0 and I_1 are two memory operations in P_i and P_j. If $I_0 \xrightarrow{d} I_1$, then $C_i(I_0) \xrightarrow{r} C_j(I_1)$.

Counting point. Pacifier counts an instruction when it is retired from the processor.

3.3.2. Relog Mechanisms Consider two instructions $I_i \xrightarrow{po} I_j$, which have both been retired. If they are reordered ($I_j \xrightarrow{r} I_i$), Relog performs the following procedure when I_i is performed.

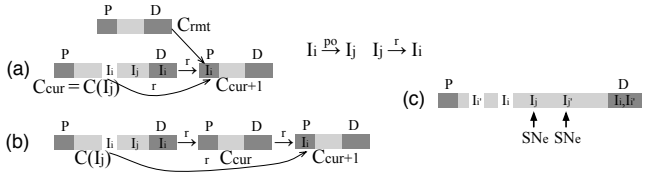

Figure 5: Relog Mechanisms.

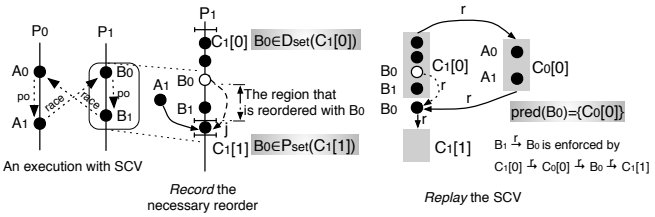

Figure 6: R&R Reordered Behavior: The Insight.

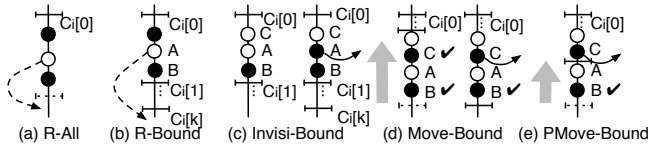

Figure 7: Minimizing recorded reorderings in SCVs.

1. The processor terminates the current chunk, C_{cur}. We have $C_{cur} = C(I_j)$ or $C(I_j) \xrightarrow{r} C_{cur}$. The second case can happen when $C(I_j)$ is terminated by another earlier event.
2. Relog records $I_j \xrightarrow{r} I_i$ by requiring $C_{cur} \xrightarrow{r} I_i$. This is recorded by logging I_i in the D_{set} of $C(I_j)$: $I_i \in D_{set}(C(I_j))$. During the replay of $C(I_j)$, I_i needs to be "skipped".
3. Let the next chunk after C_{cur} be C_{cur+1}. I_i is also logged in the P_{set} of C_{cur+1}: $I_i \in P_{set}(C_{cur+1})$. Before the replay of C_{cur+1}, I_i needs to be executed ("compensated") before C_{cur+1} starts execution.
4. If I_i has a $\xrightarrow{d}$ with an access in a remote chunk C_{rmt}, I_i needs to be replayed after C_{rmt}: $C_{rmt} \xrightarrow{r} I_i$. This order is recorded in the *predecessor chunks* of I_i: Pred(I_i) += C_{rmt}.

Figure 5 (a) illustrates the ideas. If $C(I_j) \xrightarrow{r} C_{cur}$, $I_i \in D_{set}(C(I_j))$ can still ensure $I_j \xrightarrow{r} I_i$ due to transitivity: $C(I_j) \xrightarrow{r} C_{cur} \xrightarrow{r} I_i \xrightarrow{r} C_{cur+1}$. This is shown in Figure 5 (b).

The delayed execution of stores can be implemented by the simulated store buffer, similar to the technique in CoreRacer [16]. A load in the current chunk cannot be delayed since it will prevent the chunk from finishing. To handle this, we record the load value. During replay, the recorded value overrules the value read from the memory system. This mechanism provides an "illusion" that a load in D_{set} is executed "in the future" with a return value different from the current state of the memory system. It also means that a load never needs to be executed again in the future, so P_{set} only contains stores.

If I_j is not retired, C_{cur}'s termination needs to be scheduled in the future, with the constraint $SN_e(C_{cur}) \geq SN(I_j)$. The processor can mark the chunk termination in the ROB entry of I_j and insert I_i into $D_{set}(C_{cur})$. After I_j is retired and C_{cur+1} is created, only the stores in the $D_{set}(C(I_j))$ are copied to $P_{set}(C_{cur+1})$. [2]. Before C_{cur} terminates, the processor may need to log another pair of reordered instructions, $I'_j \xrightarrow{r} I'_i$ ($I'_i \xrightarrow{po} I'_j$) If $SN(I'_j) > SN(I_j)$, the scheduled C_{cur} point needs to be adjusted to at least after I'_j with I'_i in the $D_{set}(C_{cur})$. To enable this, a processor keeps the "scheduled" SN_e of C_{cur} and ensures that any update to SN_e must be larger than the current value.

Like Rainbow [17], Pacifier does not require any knowledge about the memory model of the processor. If both I_i and I_j ($I_i \xrightarrow{po} I_j$) are loads, the reordering must happen before either of them is retired. In RC, I_j will not be re-executed and the reordering will persist, so Relog records it correctly. However, an aggressive SC implementation with in-window speculation also allows this to happen "temporarily" since I_j will be re-executed after the detection of a conflict. We handle both cases correctly as follows. If Relog observes that both I_i and I_j are loads, it conservatively logs all the load values from I_i to I_j. In the replay, all the loads in this region will use the logged values instead of the ones from the memory system. This approach is conservative because if the machine supports RC, only the value of I_i needs to be logged.

3.3.3. An Example Figure 6 shows an example. The white dot indicates the reordered (not performed) instruction. To replay the SCV, we need to break one of the two program orders (either $A_0 \xrightarrow{po} A_1$ or $B_0 \xrightarrow{po} B_1$). Suppose we break $B_0 \xrightarrow{po} B_1$ by requiring $B_1 \xrightarrow{r} B_0$. P_1 will terminate the current chunk $C_1[0]$ and create a new chunk $C_1[1]$. B_0 is inserted into $D_{set}(C_1[0])$ and $P_{set}(C_1[1])$, this captures the reordering between B_0 and all the instructions I_k, where $SN(I_k) \in (SN(B_0)+1, SN_e(C_1[0]))$. Also, B_0 should be executed after the remote access A_1, this is captured by logging A_1's chunk ($C_0[0]$) as the predecessor chunks of B_0: Pred(B_0) += $C_0[0]$.

In the replay, we have $C_1[0] \xrightarrow{r} C_0[0]$, and B_0 is skipped in $C_1[0]$. After executing $C_1[0]$, P_1 is stalled until $C_0[0]$ finishes and wakes up B_0, because $B_0 \in P_{set}(C_1[1])$ and Pred(B_0) = $C_0[0]$. Eventually, Relog ensures the correct total order of execution: $C_1[0] \xrightarrow{r} C_0[0] \xrightarrow{r} B_0 \xrightarrow{r} C_1[1]$.

In summary, Relog is a general mechanism to record the necessary reorderings in a chunk-based R&R scheme to reproduce SCVs. Naively recording all reorderings during execution is sufficient to reproduce the SCVs but is overkill and will incur large log overhead. The goal of the optimization should be *to minimize the logged reorderings but still be able to reproduce all SCVs*.

3.4. Design Space of SCV-D

R-All. The naive SCV-D is to record all reorderings locally. This is shown in Figure 7 (a). R-All obviously incurs large extra logs since the reordering is common in relaxed-consistency model or even aggressive SC.

R-Bound. A chunk is a sequence of communication-free

[2] We use the unique entry format for P_{set} and D_{set}.

instructions, so the reorderings inside a chunk are always invisible to other processors and do not need to be recorded. The reorderings only need to be recorded at chunk boundaries.

As shown in Figure 7 (b), if $C_i[0]$ needs to terminate when A and B are reordered (we assume B is retired), the chunk boundary should be at least after B: $SN_e(C_0[0]) \geq SN(B)$. Later, when A is performed, it is inserted into $D_{set}(C_i[0])$, then the current chunk (at the later point) is terminated and a new chunk $C_i[k]$ is created. There can be several chunks in between $C_i[0]$ and $C_i[k]$. A is inserted into $P_{set}(C_i[k])$.

Invisi-Bound. This optimization avoids unnecessary logging instructions in D_{set} and P_{set}. A chunk is *closed* when it is not the current chunk in a processor and a chunk is *completed* when all the instructions in it have been performed and completed. The boundary of a closed chunk is determined even if not all instructions inside it have been performed.

With Invisi-Bound, when an instruction I_k in a closed chunk $C(I_k)$ is performed, if it is not involved in any dependence, then I_k does not have to be inserted to $D_{set}(C(I_k))$ or P_{set} of a new chunk. The insight is that the reordering of I_k and the instructions in $C(I_k)$ after I_k are not visible to other processors. Therefore, I_k can be executed together with the other instructions in the closed chunk. Note that when an instruction is performed, it can only be the destination of a dependence.

Only a performed instruction can be the source of a dependence. When a performed instruction I_k in a closed but not completed chunk $C(I_k)$ becomes the source of a dependence, the current chunk (the only one that is not closed) of the processor C_{cur}, is ordered before the remote chunk (C_{rmt}) that contains the destination of the dependence. In this case, only $C_{cur} \xrightarrow{r} C_{rmt}$ is recorded. Even if there is no direct order from $C(I_k)$ to C_{rmt}, this order is still enforced by transitivity: $C(I_k) \xrightarrow{r} C_{cur} \xrightarrow{r} C_{rmt}$. We can still ensure that all instructions (including I_k) in $C(I_k)$ are ordered before C_{rmt}. Therefore, in Invisi-Bound, counting I_k in $C(I_k)$ is correct.

Figure 7 (c) gives an example. $C_i[0]$ initially contains two instructions not performed (C and A). When A is performed without incurring a dependence, Invisi-Bound counts it as a part of $C_i[0]$. Later, before $C_i[0]$ is completed, A may become the source of a dependence. The correct order is ensured by $C_i[0] \xrightarrow{r} C_i[k] \xrightarrow{r} C_{rmt}$, where $C_i[k]$ is the current chunk and C_{rmt} is not indicated.

Move-Bound. This optimization checks whether any of the instructions in PW are the source of any dependences when C_{cur} needs to terminate. If not, the recorder can decide to *move* the boundary up to the position before the oldest instruction in PW. After the adjustment, no new chunk needs to be created and no instructions need to be logged in D_{set} and P_{set}. Essentially, Move-Bound "delays" the decision on whether the reordering needs to be recorded by changing the chunk boundary so that the potentially reordered instructions are still a part of the current chunk. Eventually, they may be safely included in the current chunk without logging the reordering information (enabled by Invisi-Bound). Without Move-Bound, even if the reordering of the not-yet-performed instructions are eventually not visible, we have to log them (unnecessarily) just because of the chunk boundary. Move-Bound is correct because the potential reorderings of the not performed instructions have not been made visible to other processors, so adjusting the chunk boundary across them is safe.

Opt.	Observation	Benefit
R-Bound	Chunk is communication free.	Only record reordering at chunk boundary
Invisi-Bound	Some reorderings in closed chunk are invisible.	No unnecessary reordering logging in closed chunk.
Move-Bound	All reorderings in PW are invisible.	(1) No artificial reordering. (2) More precise SCV-D (see Section 5.2 and Figure 10).
PMove-Bound	Some of the reorderings in PW are invisible.	

Table 2: Reducing Recorded Reordering: Optimization Space.

Figure 7 (d) illustrates Move-Bound. On the left, the current chunk $C_i[0]$ has two performed instructions (C and B). If termination of $C_i[0]$ is required, the boundary would have been at least after B. With Move-Bound, since C and B are not the sources of any dependences (marked as ✓), the chunk boundary can be moved up to the point before the entire PW, which keeps all four instructions in the current chunk without creating a new one. For the case on right of Figure 7 (d), if such a move is applied, no ordering requirement can capture $C \xrightarrow{r} C_{rmt}$. Without the move, it is captured by $C_i[0] \xrightarrow{r} C_{rmt}$.

PMove-Bound. This optimization is a generalization of Move-Bound. Different from Move-Bound, PMove-Bound enables the "partial" move of the chunk boundary up to the first retired instruction that is the source of a dependence.

The idea is shown in Figure 7 (e). When $C_i[0]$ needs to terminate, C is performed and is the source of a dependence. PMove-Bound still allows the boundary to move right after C. Such partial move is valid because the reordering between A and B is still invisible.

The optimizations are compared in Table 2. The strength and benefits progressively increase. The optimizations presented are orthogonal to chunk termination policy.

3.5. Granule: A Practical and Lightweight SCV-D

SCV is a pattern that requires at least two cyclic data races overlapped in a special way [18]. R&R schemes that only terminate chunks on cyclic dependences (e.g., Karma [3]) are *inherently* superior to ones that terminate the chunk at the first dependence (e.g., Rerun [9]). The former category naturally captures a *subset* of the SCV properties. To design an efficient SCV-D, it is natural to start with Karma.

This section presents the design of *Granule*, an SCV-D using the chunk termination policy in Karma [3] with PMove-Bound and Invisi-Bound(Karma + PMove-Bound+ Invisi-Bound). We also discuss two alternative designs:

- Karma + Move-Bound+ Invisi-Bound. This combination shows the additional benefits of PMove-Bound.

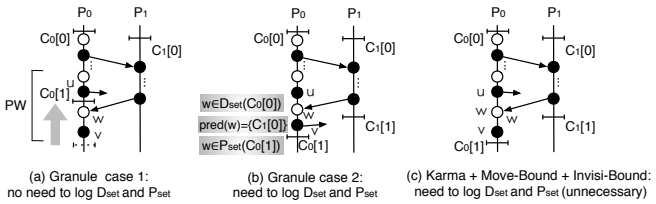

Figure 8: R&R SCV-D Designs.

- A hypothetical design using Volition [18], a precise SCV detector as SCV-D. This is to show why Granule is a good approximation of Volition with much lower overhead.

While Granule is based on Karma, this scheme by itself *cannot* be directly used as an SCV-D because it does not have the policy to (1) record all out-of-order execution behavior and (2) decide what to do with the reordered instructions at the chunk boundary.

3.5.1. Granule: Karma + PMove-Bound+ Invisi-Bound Consider Figure 8 (a). When the first dependence between two processors is formed, there is no need to record any reordering, because there is no chunk termination request. When the second dependence (from an access in P_1 to *w*) occurs, Karma requires that $C_0[0]$ should terminate. The new chunk $C_0[1]$ has to start at least after the latest retired instruction *v*. According to PMove-Bound, the boundary of $C_0[0]$ can be moved right after *u* since *v* is not a source of a dependence. The reordering between *w* and *v* is not visible so there is no need to log any extra information in D_{set} and P_{set}.

When $C_0[1]$ is created, $C_0[0]$ is closed but may not have completed (e.g., *u* may not have been performed). According to Invisi-Bound, at a later point when an instruction in $C_0[0]$ is performed (e.g., *u*), if it is not the destination of any dependence, then there is no need to log any reordering either.

Instead, if *u* is indeed a destination of a dependence when performed later, Karma will initiate another chunk termination request, which could be for $C_0[1]$ or later ones in P_0. Now because $C_0[0]$ is closed, the "destiny" of *u* has been decided — it can only be placed in $C_0[0]$. Therefore, the reordering needs to be logged. In particular, *u* will be inserted into $D_{set}(C_0[0])$ and $P_{set}(C_0[1])$ or P_{set} of a later chunk.

Figure 8 (b) shows a similar scenario when *u* is performed, the only difference is that *v* is the source of a dependence. According to PMove-Bound, $C_0[0]$ can only terminate at a position after *v*. Therefore, the reordering needs to be logged. Pred(*w*) indicates that *w* (and also $C_0[1]$) can only execute after the remote chunk $C_1[0]$ completes.

3.5.2. Karma + Move-Bound+ Invisi-Bound If we use Move-Bound instead of PMove-Bound, some reorderings could be logged unnecessarily. Consider Figure 8 (c). According to Move-Bound, if any instruction in PW is the source of a dependence, the chunk boundary has to be placed after the most recent retired instruction, *v*. Consequently, extra information has to be logged for *w* in the same way as in Figure 8 (b), which is unnecessary.

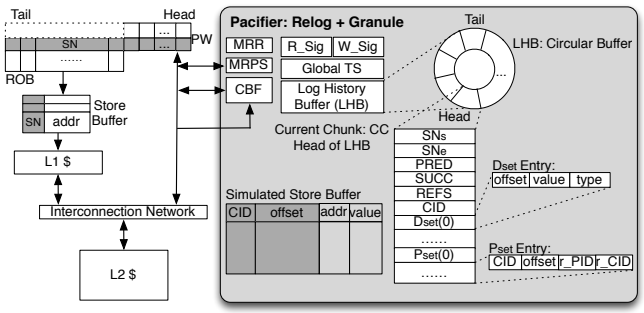

Figure 9: R&R The Implementation of Pacifier.

3.5.3. Volition as SCV-D At an extreme, one can use Volition [18], a precise SCV detector as the SCV-D module. We call this a "hypothetical" design because some details need to be considered to combine Volition with Karma, and we will not attempt to do so in this paper. The point is that Volition can provide the *minimum* set of reorderings that must be recorded to reproduce the SCVs. Therefore, Volition can serve as the "oracle".

The basic idea of Volition is shown in Figure 10 (a). It detects SCVs precisely among an arbitrary number of processors. The cycles are detected by continuously propagating dependences. A processor only propagates a dependence (*pred*) through another (*succ*) if the SN of the destination of *pred* is smaller than the SN of the source of *succ*. Based on Volition, when an SCV is detected, only one reordering needs to be logged in one of the processors in the cycle.

4. The Implementation of Pacifier

Figure 9 shows the hardware components of Pacifier. Listing 1 provides a complete and detailed description of Pacifier. Since R&R is related to the correctness, similar to Karma [3], C-like pseudocode is used to eliminate ambiguities. Due to space, we do not include the logic handling non-atomic writes.

4.1. Hardware Components

Pending Window (PW). PW is organized as a circular queue. The PW entry for each instruction is inserted in program order. **updatePW(..)** (*l*:57) shows the algorithm to delete the PW entries. The deletion of one PW entry may trigger the deletion of a sequence of PW entries that follow it. This process stops when the next instruction is not performed or PW is empty.

Log History Buffer (LHB). LHB keeps information about all the chunks that are not completed and those that are the completed but with older not completed chunks. Its structure and operation are similar to LHB. **updateLHB(..)**(*l*:63) is the delete operation.

Log Formats. Each LHB log contains (1) the basic fields in Karma and (2) D_{set} and P_{set} with combined and unique format (**PD_Set** (*l*:1)). For an open (the current) chunk, SN_e keeps the current "scheduled" chunk termination point. For closed chunks, SN_e is fixed. (r_PID,r_CID) indicates Pred. PendList is the instructions not performed in a chunk.

```
1  typedef LHB{                  typedef PD_Set{
2   long long SN_s;               long long CID;
3   long long SN_e;               int offset;
4   long long PRED;               int r_PID;
5   long long SUCC;               long long r_CID;
6   long long CID;                long long value;
7   int REFS;                     bool type;
8   std::set<PDSet_t> D_set;    } PDSet_t;   typedef PW{
9   std::set<PDSet_t> P_set;                  long long SN;
10  std::set<PDSet_t> incomp_P_set;           long long addr;
11  list_t PendList;                          bool performed;
12 } LHB_t;                                  } PW_t;
13 long long MRPS,MRR; LHB_t *cc;
14 PW_t PW[PW_SIZE]; LHB_t LHB[LHB_SIZE];
15 int PW_head,PW_tail,LHB_head,LHB_tail;
16
17 void OnChunkTerminate(long long SN, bool cycle,
18         int type,int r_PID,long long r_CID){
19  int LHB_idx=get_LHB_index(SN);
20  int offset=SN-LHB[LHB_idx].SN_s;
21  if(cycle==false){ //no cycle, just terminate
22   cc->SN_e=(MRR>cc->SN_e)?MRR+1,cc->SN_e; }else{ // cycle
23   cc->SN_e=(MRPS!=0)?MRPS+1,(SN-1);
24   if(SN<MRPS){ //Granule detects SCV
25    LHB[LHB_idx].D_set.insert(LHB[LHB_idx].CID,offset,\
26                              r_PID,r_CID,get_value(SN),type);
27    if(performed(SN)){
28     if(in_incomp_P_set(SN))
29      insert_set(cc->D_set,cc->incomp_P_set,SN);
30      clear_set(cc->incomp_P_set,SN);
31    }else{
32     cc->incomp_P_set.insert(LHB[LHB_idx].CID,offset, \
33                                r_PID,r_CID);
34    }
35   }
36  }
37  if((cc->SN_e<=MRR)&&(cc->SN_e!=0))  init_new_chunk();
38  if(cycle) send_insert_succ_message(r_PID,r_CID,my_PID);
39  if(performed(SN)) markPerformed(SN);
40 }
41
42 void init_new_chunk(){
43  LHB_head++; int h_idx=LHB_head%LHB_SIZE;
44  copy_set(LHB[h_idx].P_set,cc->D_set); //only copy stores
45  copy_set(LHB[h_idx].incomp_P_set,cc->incomp_P_set);
46  cc=&LHB[h_idx];
47 }
48
49 void instRetire(Inst_t *inst){
50  ...; MRR=inst->SN;
51  if(inst->SN==cur_chunk->SN_e) init_new_chunk();
52 }
53 void markPerformed(long long SN){
54  updatePW(SN); updateLHB(SN);
55 }
56
57 void updatePW(long long SN){
58  int idx=get_PW_index(SN); PW[idx].performed=true;
59  if(idx==PW_tail)
60   do{ idx=(PW_tail+1)%PW_SIZE; PW_tail=idx;
61   }while((PW[idx].performed)&&(idx<=PW_head));
62 }
63 void updateLHB(long long SN){
64  int idx=get_LHB_index(SN); LHB[idx].PendList.remove(SN);
65  if((idx==LHB_tail)&&(LHB[idx].PendList.empty()))
66   do{ outputLog(idx); //record closed log
67    index=(LHB_tail+1)%LHB_SIZE; LHB_tail=idx;
68   }while((LHB[idx].PendList.empty())&&(idx<=LHB_head));
69 }
70
71 void replay(LHB_t chunk){
72  execute pending set first;
73  while(not all instructions in P_set are performed) {};
74  while(chunk.REFS!=0){
75   Inst *cur_i=getInst(chunk);
76   if(inDSet(cur_i))
77    if((isLoad(cur_i))){ Use the value in D_set; }
78     else{ Insert_Sim_SB(cur_i); }
79     else{ Execute cur_i; }
80     chunk.REFS--; } //while
81 }
```

Listing 1: Record and Replay Algorithm of Pacifier

Simulated Store Buffer (SSB). SSB is used to replay the delayed stores. The stores in D_{set} are held in SSB. When a delayed store is ready to execute (found in a P_{set}), it is identified by (CID, offset) from the SSB and executed.

Counting Bloom Filter (CBF). CBF summarizes all the addresses in PW to avoid unnecessary associative search when a processor checks the condition for PMove-Bound.

Most Recent Pending Source (MRPS). MRPS is a register indicating the SN of the youngest performed instruction in PW that is the source of a dependence. It is used in PMove-Bound to indicate a "earliest point" that a new chunk can start from.

Most Recent Retired (MRR). MRR is a register indicating the SN of the most recently retired instruction.

4.2. Record Algorithm

OnChunkTerminate(..)(l:17) is invoked when the current chunk termination request is generated. "cycle" indicates whether the termination is due to a cycle. "SN" is the SN of the destination access of the dependence that closes the cycle (*dinst*). "Type" indicates whether this is a load (0) or store (1).

In **OnChunkTerminate(..)**, if there is a cycle (l:23-35), we decide the termination point in l:23. If no instruction in PW is the source of a dependence (MRPS == 0), *cc* (current chunk) terminates at (SN-1). Otherwise, the new chunk at least starts from (MRPS+1). If the SN of *dinst* is smaller than *MRPS*, Granule detects an SCV (l:25). Then the information of *dinst* and the dependence are recorded in the D_{set} of *cc*. If *dinst* is a load, the value (**get_value(SN)**) is recorded (l:25).

If *dinst* is a store, before it is globally performed, the potential reordering intermediate information caused by WARs is recorded in *incomp_P_Set* (l:32). When the store is globally performed, the entries for this store in incomp_P_{set} are copied to D_{set} of *cc* (l:29) and P_{set} of the next chunk (l:45). [3] This means that for the stores that incur multiple WARs, they will be replayed after all the chunks that received the invalidations.

In the end, if SN_e is smaller than MRR, a new chunk is created after MRR (l:37). Then the remote source processor is notified to set up the successor so that the instructions in the pending set of the new local chunk can be properly woken up in the replay (l:38). Finally, when *dinst* is performed, delete operations on PW and LHB (l:39) are triggered.

init_new_chunk() (l:42) creates a new chunk. The entries for *stores* in D_{set} of the current chunk are copied to the P_{set} of the new chunk (l:44). Also, *incomp_P_Set* of the current chunk is passed to the next chunk, since those stores have not been globally performed (l:45).

4.3. Replay Algorithm

replay(LHB_t chunk) (l:71) shows the replay operations in Pacifier. The instructions in P_{set} need to be performed before

[3] This is for an atomic write. For a non-atomic write in P_i, if it is performed w.r.t. P_j, which also observes the new value, then a future WAR from another processor P_k will not incur a new order from P_k to P_i. Instead, P_i will request that P_k log the load value. See Section 3.2 for details.

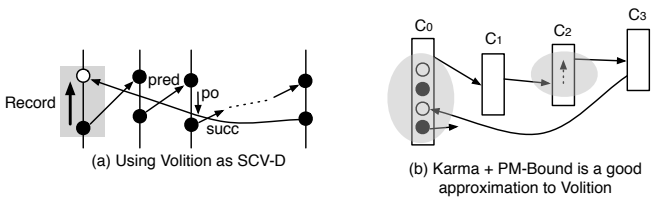

Figure 10: R&R Comparing Volition and Granule.

the other instructions are executed (*l*:72-73). The loads in D_{set} use the values in the log (*l*:77) and the stores in D_{set} are inserted into the SSB (*l*:78). The stores in SSB will be later executed just before later chunks which includes them in P_{set}.

5. Discussion

5.1. Comparing RelaxReplay and Pacifier

RelaxReplay [8] (independently developed) is the first R&R scheme that solves $R^3M^2D^2$ assuming write atomicity. There are three distinctions between RelaxReplay and Pacifier.

SCV-D Precision. RelaxReplay uses local checks to detect the SCVs, so it can be naturally applied to the distributed directory. However, such local checks come at a price. While Opt SCV-D avoids some false positives in Base SCV-D, in essence it *uses data races as proxies for SCV*. Similar techniques are used in post-retirement speculation (e.g., InvisiFence [4]), where the processor checks the conflicts (races) between the incoming coherence transactions and the local speculatively executed instructions. On conflicts, the processor rolls back and re-executes a sequence of instructions. In Opt SCV-D, such conflicts indicate that the local reorderings are visible and the reordered instructions are logged. However, many data races *do not* lead to SCVs [13, 18]. Therefore, even Opt SCV-D can incur false positives, which translate to unnecessary logs. The SCV-D of Rainbow [17] is more precise than RelaxReplay since an SCV is not reported on the first conflict.

The SCV-D of Pacifier, Granule, is based on Karma [3]. Its chunk termination policy inherently captures an important feature of SCVs (i.e., cyclic dependence). If a processor just logs all the reorderings according to Relog when a chunk needs to terminate, it is likely to generate fewer logs than RelaxReplay, because this only happens on cyclic dependences. In addition, Granule augments it with both Invisi-Bound and PMove-Bound, further reducing false positives.

Non-atomic writes. RelaxReplay *cannot* support non-atomic writes. The reason is that it assumes *a single performed point for each store*. Referring to Figure 2 (d), the global performed point of (1) is after (5) in P_2. Therefore, it is possible that (1) is counted (globally performed) right after (5) $\xrightarrow{d}$ (1) is formed, then we have CISN == PISN, which indicates that there is no need to record any reordered instruction even with Base SCV-D. Unfortunately, this is a real SCV that has to be recorded. It is unclear how RelaxReplay can be extended, since there is no notion of multiple performed points.

Pacifier can naturally capture multiple performed points because it monitors *dependences* from a remote chunk to a local access. A single write can incur multiple WAR dependences, which correspond to the multiple performed points.

Replay Style. Both RelaxReplay and Pacifier use the insights from Rainbow [17] to record the reordered instructions. Interestingly, the record style in RelaxReplay is "complementary" to the style in Rainbow and Pacifier. Instead of logging an instruction that should be delayed (in D_{set}) and scheduling it to execute later (in P_{set}), RelaxReplay places *a reordered* instruction to an *earlier* interval and leaves a dummy entry in the interval that the instruction is counted. Such different styles are due to the definition of interval in RelaxReplay and its decision to count an instruction when it completes (removed from PW) instead of when it retires from processor.

5.2. Comparing Granule and Volition

We compare the high level operation of Granule and Volition shown in Figure 10 (a) and (b), respectively. We can easily see the similarities, which include: (1) Both propagate dependences; (2) Both detect cyclic dependences among an arbitrary number of processors; (3) When a cycle is detected, both check whether the destination of the dependence that closes the cycle is older than any instruction in PW that is the source of a dependence.

Among these similarities, (3) is in particular the key reason why Granule can detect SCVs with high precision. It is marked by the shaded area in C_0 (compared with the shaded square in the first processor in Figure 10 (a)). Intuitively, this is a relatively particular pattern that does not happen easily, but it captures one of the essential properties of SCV: The ability to distinguish the special pattern attributes to PMove-Bound. Therefore, we see that Granule is able to identify relatively specific SCV patterns and only record the necessary reorderings for them.

Granule is different from Volition in multiple aspects (see Table 3). Those factors make Granule an approximative SCV-D but reduce the hardware overhead significantly. We believe Granule is a sweet spot in the design space of SCV-D since it represents a reasonable trade-off between hardware and detection overhead.

6. Evaluation

6.1. Evaluation Setup

We implement Pacifier in the SESC [19] cycle-level architectural simulator. We model a multicore with 16, 32 or 64 processors and a MESI directory-based cache coherence protocol with Release Consistency (RC). We do not model non-atomic writes. We compare Pacifier with an oracle configuration using Volition [18] as SCV-D. They are marked as *Gra* and *Vol* in the plots. We also report the results of Karma [3]. With RC, Karma is merely used to compare the overhead. The configuration of the simulated machine is shown in Table 4. We execute ten applications from the SPLASH-2 suite.

	Volition [18]	*Granule*	Overhead Reduction
Multi-Word Line	Keep extra states for each word of a cache line.	Track dependences at cache line level.	No extra per-word state.
Cycle Detection	Precise. Propagate all dependences.	Imprecise. Use simple scalar timestamp [10].	Fewer messages for dep. propagation
Race Propagation Condition	See Figure 8 (d). Propagate only when: SN(Dst(pred)) $\leq$ SN(Src(succ)), "Dst" and "Src" are the destination and source of a dep.	Propagate on any dep. between two chunks, even if po cannot "connect" the two (see shaded area in C_2 in Figure 8 (e))	No local logic to check whether a propagation should happen.
Race Clearance	Remove a race when the source is not in PW.	Never remove the order between chunks.	No messages to notify the removal.
SCV Detection	Precise.	Imprecise due to the above factors.	—

Table 3: Comparison of Volition and Granule.

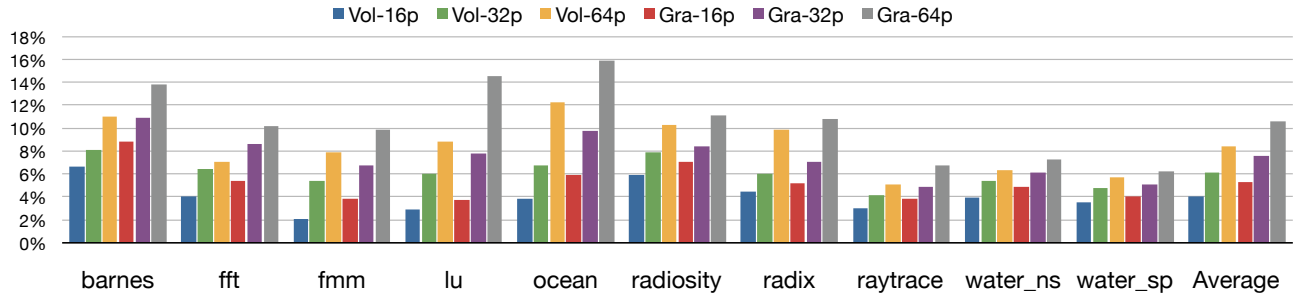

Figure 11: Pacifier Log Size.

Architecture	Multicore chip with 16, 32 or 64 cores
Core width; ROB size	4-issue; 128 entries
Consistency	RC
Store buffer	32 entries with randomly added delays between 0 to 50 cycles
Private L1 cache	32KB WB, 4-way, 2-cycle round trip
Shared L2 cache	1MB module/proc. Module: WB, 8-way
Latency to L2	Local module: 11-cycle round trip
Cache line size	32 bytes
Cache coherence	Directory-based MESI protocol
Network	2-D mesh with 7-cycle hop latency
Main memory	200-cycle round trip
Pacifier parameters	PW size: 256 entries; Number of LHBs per core: 16;

Table 4: Architecture parameters.

6.2. Log Size Comparison

Figure 11 compares the log sizes with different number of processors using either Volition or Granule as the SCV-D component. We show the percentage of log size increase compared to Karma with the same number of processors. The log size increase is due to the extra information logged in Relog (D_{set} and P_{set}). We see that for each application, as the number of processors increases, the log sizes increase faster, implying that a larger number of processors is more likely to incur SCVs. Over all experiments, Pacifier (Gra) incurs 3.9% $\sim$ 16% extra logs due to SCVs. On overage, the log size increases in Vol for 16, 32 and 64 processors by 4.1%, 6.1% and 8.5%, respectively. The increases in Gra for 16, 32 and 64 processors are 5.3%, 7.6% and 10.7%, respectively. We can see that Granule is a very good approximation of Volition.

6.3. Replay Speed Comparison

Figure 12 shows the percentage of replay slowdown using Karma, Vol and Gra, compared to native execution. The bars for the same number of processors are normalized to the native execution time. We see that the original Karma does achieve a good replay speed. As the number of processors increases, the replay speed tends to be lower compared to the native execution. This is due to the longer latency to wake up the remote chunks. Reproducing the SCVs can further slow down the replay. Over all experiments, the slowdown of Pacifier (Gra) in the replay is 10.1% $\sim$ 30.5% compared to native execution. On average, the slowdowns of Karma in replay with 16,32 and 64 processors are 10.3%, 13.6% and 19.1%, respectively. For Vol, the slowdowns are 11.3%, 14.7% and 21.4%, respectively. For Gra, the slowdowns are 12.4%, 16.1% and 23.4%, respectively.

6.4. LHB requirement

Figure 13 reports the maximum number of LHB entries occupied in the different executions using 16, 32 and 64 processors with Vol and Gra. We see that the requirement of LHB is very modest: with 16 LHBs configured, the largest number of used CLUs observed is 7 (radiosity with Vol). It is worth noting that the results are based on the configuration with a randomized delay between 0 to 50 cycles for the stores in the store buffer, which can potentially increase LHB utilization. We expect that in reality, with a reasonable number of LHBs, the chance of stalling the execution due to LHB overflow is extremely low.

7. Conclusion

This paper proposes *Pacifier*, the first chunk-based solution to $R^3M^2D^2$ without assuming write atomicity. Pacifier is composed of *Relog* and *Granule*. Relog is a general memory reordering logging and replay mechanism that can reproduce SCVs in relaxed memory models. Granule is an SCV detection scheme in the record phase with good precision, indicating when to record with Relog. We show that Pacifier is a sweet spot in the whole design space because it represents a reason-

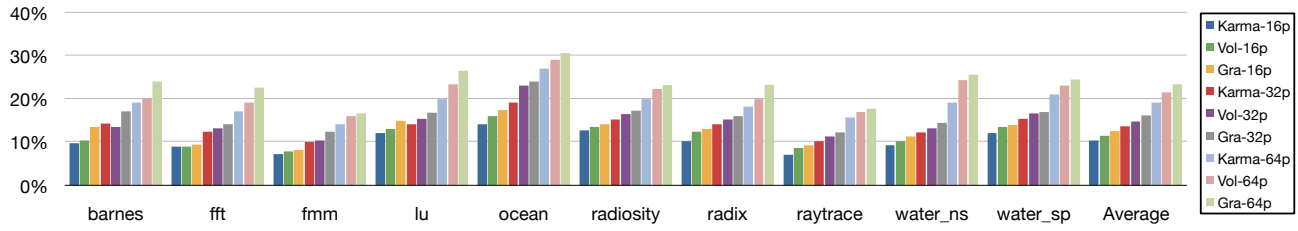

Figure 12: Pacifier Replay Speed.

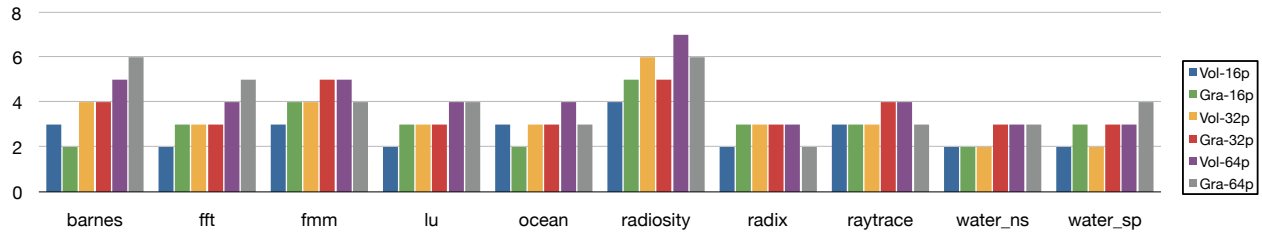

Figure 13: Utilization of LHB.

able trade-off between hardware cost and detection precision. An evaluation with simulations of 16, 32 and 64 processors with Release Consistency (RC) running SPLASH-2 applications indicates that Pacifier incurs 3.9% $\sim$ 16% larger logs due to SCVs. The slowdown of Pacifier in the replay is 10.1% $\sim$ 30.5% compared to native execution.

Acknowledgements

We thank Hongbo Rong, Joel Galenson, Michael Pradel, Phitchaya Mangpo Phothilimthana, the anonymous referees and our shepherd, Sandhya Dwarkadas, for their invaluable feedbacks and suggestions that helped improve the paper.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[2] ARM. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition Issue C. 2012.

[3] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. Karma: Scalable Deterministic Record-Replay. In *International Conference on Supercomputing*, 2011.

[4] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. Invisifence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *International Symposium on Computer Architecture*, 2009.

[5] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. LReplay: A Pending Period based Deterministic Replay Scheme. In *International Symposium on Computer Architecture*, 2010.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, 1990.

[7] Nima Honarmand, Nathan Dautenhahn, Josep Torrellas, Samuel King, Gilles Pokam, and Cristiano Pereira. Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism . In *International conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[8] Nima Honarmand and Josep Torrellas. RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[9] Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *International Symposium on Computer Architecture*, 2008.

[10] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[11] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, July 1979.

[12] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*, 2008.

[13] Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *International Symposium on Microarchitecture*, December 2012.

[14] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording Shared Memory Dependencies Using Strata. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[15] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *International Symposium on Microarchitecture*, 2009.

[16] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. CoreRacer: A Practical Memory Race Recorder for Multicore X86 TSO Processors. In *International Symposium on Microarchitecture*, 2011.

[17] Xuehai Qian, He Huang, Benjamin Sahelices, and Depei Qian. Rainbow: Efficient Memory Race Recording with High Replay Parallelism for Relaxed Memory Model. In *International Symposium on High Performance Computer Architecture*, 2013.

[18] Xuehai Qian, Benjamin Sahelices, Josep Torrellas, and Depei Qian. Volition: Scalable and Precise Sequential Cons istency Violation Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[19] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[20] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[21] D. J Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[22] Gwendolyn Voskuilen, Faraz Ahmad, and T. N. Vijaykumar. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *International Symposium on Computer Architecture*, 2010.

[23] Min Xu, Ratislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *International Symposium on Computer Architecture*, 2003.

[24] Min Xu, Ratislav Bodik, and Mark D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.