Recording Shared Memory Dependencies Using Strata

Satish Narayanasamy[†] Cristiano Pereira[†] Brad Calder[†‡]

[†]University of California - San Diego [‡]Microsoft

Abstract

Significant time is spent by companies trying to reproduce and fix bugs. BugNet and FDR are recent architecture proposals that provide architecture support for deterministic replay debugging. They focus on continuously recording information about the program's execution, which can be communicated back to the developer. Using that information, the developer can deterministically replay the program's execution to reproduce and fix the bugs.

In this paper, we propose using *Strata* to efficiently capture the shared memory dependencies. A stratum creates a time layer across all the logs for the running threads, which separates all the memory operations executed before and after the stratum. A strata log allows us to determine all the shared memory dependencies during replay and thereby supports deterministic replay debugging for multi-threaded programs.

Categories and Subject Descriptors C. Computer Systems Organization [*C.1 Processor Architectures*]: C.1.4 Parallel Architectures

General Terms Design, Measurement, Performance, Reliability

Keywords Strata, Replay, Debugging, Logging, Shared Memory Dependencies

1. Introduction

Hardware techniques have been proposed to continuously record the program's execution with very little overhead (around 1%) [13, 6] to assist developers by supporting *Deterministic Replay Debugging* (DRD). Deterministic Replay Debugging enables a programmer to replay the exact same sequence of instructions that led up to the crash, and therefore it is an effective technique to understand the source of the bug. The hardware techniques can support deterministic replay of the last second of execution preceding the crash, which was found to be sufficient to debug the root cause of the bug [6]. Since the overhead of these hardware techniques are low enough, they are transparent and hence they can always be left on during production runs.

One of those techniques is called the Flight Data Recorder (FDR) [13]. FDR creates checkpoints based on SafeyNet [11] to support full system deterministic replay. Another approach is called BugNet [6]. BugNet logs the load values executed by the application and supports deterministic replay of the application code and shared libraries, as opposed to the full system replay supported by FDR.

For debugging multi-threaded programs, the techniques described above need an efficient mechanism to record the shared memory dependencies between the threads. To accomplish this, FDR [13] logs the shared memory dependencies in a *Memory Race Log*, which is maintained for each processor node. FDR determines the shared memory dependencies for a thread by monitoring its coherence messages. In order to reduce the size of the memory race logs, FDR implemented the Netzer optimization in hardware [7]. BugNet [6] assumed the same method for recording the shared memory dependencies in its memory race logs. We refer to the logging method used by FDR and BugNet as the *point-to-point* logging approach, because to capture a dependency, they log the instruction counts of both the dependent operations.

In this paper, we propose capturing the shared memory dependencies using *Strata*. A stratum is logged when a shared memory dependency needs to be captured. It consists of the memory counts of all the threads at the time when it is logged. A stratum separates all the memory operations that were executed in all the threads before the time when it is recorded, from those that will be executed after it is recorded. Since the stratum is recorded just before the execution of the dependent memory operation, the stratum separates that memory operation from the earlier memory operation in which it is dependent on.

The benefits of using strata are (1) it enables us to design a hardware solution for logging shared memory dependencies in both snoop-based and directory based systems, whereas the previous point-to-point logging solution only supported directory based systems, (2) the strata logging approach does not require us to log the shared memory write-after-read (WAR) dependencies, which can be determined during replay, (3) a single stratum can capture many different dependencies and as a result the strata logging approach reduces the number of memory dependencies logged even more than the prior Netzer optimization for point-to-point logging solution, and (4) the hardware required to create the strata log is smaller than what is required for implementing the point-to-point logging solution.

2. Prior Work

In this section we summarize the prior techniques proposed to capture shared memory dependencies.

One of the first hardware support for deterministic replay for a program executing in a multi-processor system was proposed by Bacon and Goldstein [1]. Their design was for a bus based system. They observed that dependencies between the threads executing in a multi-processor system can be captured by monitoring the coherence messages on the bus. However, they recorded all the coherence traffic on the bus, which can result in a large log size. Also, their

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

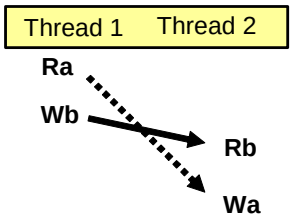

Figure 1. Netzer Transitive Optimization.

system cannot handle non-determinism due to the system interactions.

The amount of information that needs to be logged to record the memory access ordering can be reduced by applying the Netzer's transitive optimization [7]. FDR [13] proposed a design to implement the Netzer transitive optimization in a directory based system assuming sequential consistency. FDR adopts the SafetyNet [11] checkpoint mechanism to retrieve a consistent full system state corresponding to a prior instance in time. Additionally, it records all the inputs coming into the system (I/O, interrupts, DMA transfers) to enable replaying. With this recorded information, starting from the retrieved full system state, the original program execution can be replayed.

BugNet [6] focused on supporting deterministic replay of the user code and the shared level libraries by recording the output of load instructions. Thus, it can support application level debugging. It can handle all forms of non-determinism due to system interactions and shared memory updates. To capture the shared memory dependencies, it implemented FDR's point-to-point logging.

ReEnact provides an approach for rolling back and replaying the execution using thread level speculation support [9]. Its main goal is to dynamically detect data races and it does not support deterministic replay of non-deterministic I/O interactions. CORD [8] also captures some, but not all, of the RAW dependencies in a snoop based system in order to detect data races.

In the software based InstantReplay [3] system, accesses to the shared objects are forced to go through a procedure, which logs information about the shared access to the object in a log. InstantReplay is suitable for systems where the objects are shared at a coarser level (through monitors and message queues). However, performance degradation when the program uses fine-grained shared memory accesses can be severe. To support replay of multi-threaded programs on a uni-processor system, DejaVu [2] proposed recording just the scheduler decisions.

3. Baseline: The Point to Point Approach to Log Shared Memory Dependences

To replay and debug multi-threaded applications we need to record the memory dependencies that exist across all the threads. To accomplish this, the prior techniques used the point-to-point logging approach proposed by Flight Data Recorder(FDR) [13]. In this section, we discuss FDR's algorithm and its hardware design to capture the shared memory dependencies.

3.1 Point-to-Point Logging and Netzer Optimization

FDR captures all forms of shared memory dependencies: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) dependencies. These dependencies are logged in *Memory Race Logs*. Logging every dependency seen during execution is impractical as it would lead to unmanageable memory race log sizes.

In order to reduce the log sizes, FDR implemented the Netzer optimization [7] in hardware. FDR's hardware design assumed a directory based system that implemented a sequentially consistent memory model. We briefly explain the Netzer algorithm using a simple example.

The Netzer algorithm works by exploiting the transitive property in a system that assumes sequential consistency. A simple example for two threads, T1 and T2, is shown in Figure 1, where each thread executes a write and a read. The subscripts represent the address locations.

FDR records the dependency $Wb \rightarrow Rb$ in a memory race log. The dependency between the two threads is recorded using the two instruction counts of the dependent threads, which is a form of time-stamp used in FDR. This is sufficient because all that we need to know while replaying is that T1 should have been executed *at least* until the memory operation *Wb*, before T2 can execute its memory operation *Rb*. Later, when we observe the second dependency $Ra \rightarrow Wa$ between T1 and T2, it does not have to be recorded because it is transitively implied by the previously recorded dependency.

We call the logging approach used in FDR as the *point-to-point* logging approach, because each dependency is logged by explicitly logging the instruction counts of the two dependent memory operations executed in two different threads.

3.2 Hardware support for Point-to-Point Logging

In order to capture all the shared memory dependencies between the threads, we should first be able to detect them when the program is executing. FDR [13] records shared memory dependencies between the processor nodes (and not the threads). We believe that this is sufficient information, because we can map the recorded dependencies between processor nodes back to the threads during replay. This requires that we know which thread was executing on a processor node at a given time. This is required in BugNet, as it can replay only user level code, and hence cannot reproduce the thread scheduling orchestrated by the operating system.

Intra-node dependencies (dependencies within the same processor node) need not be logged as they are trivially revealed by the program order. To detect dependencies between the processor nodes, FDR uses an observation that, those dependencies are revealed by the coherence messages. There can be a cross-node shared memory dependency (dependency between two different processor nodes), when a processor node encounters a read/write cache miss. If there are processor nodes in the system that have a read/write permission, then the appropriate dependency (RAW or WAW or WAR) with those processor nodes can be detected. If none of the processor nodes in the system have read/write permission for the memory block (that is, the block is not cached anywhere), then the directory entry has information about the last writer to the memory block. If the last writer is different from the processor node that generated the read/write miss, then a cross-node dependency is detected. In the MESI directory protocol, when a clean block is evicted, the directory is not informed (silent eviction). Therefore, the directory entry continues to contain information about the readers in the system (sharers) till there is a write in any of the processor nodes. Thus, the cross-node WAR dependencies can be detected for a processor node that read a block and evicted it.

However, for systems based on snoopy protocol there is no directory to hold the last writer information and the list of readers for the blocks that have been evicted from the cache. Hence, additional support is required in snoop-based systems, and this is not solved in the prior proposals [13, 6]. In Section 5, we describe how our strata logging approach can be easily implemented for snoop-based systems.

For directory based system, there is still a corner case which was not addressed in the prior FDR and BugNet proposals [13, 6]; one that is related to paging. If a physical page is swapped out, then future accesses to that page will not find correct shared memory dependencies as the directory loses the information about the processor nodes that last accessed the paged out memory blocks. This is not a problem in our strata logging approach, which will be explained in detail in Section 4.3.

3.2.1 Hardware support for Implementing Netzer Algorithm

Going back to our example, assume that FDR [13], by observing the cache coherence messages, detects the dependency *Wb→Rb* between the two processors P1 and P2 executing the threads T1 and T2 respectively (the threads are shown in Figure 1). This dependency is recorded in the *Memory Race Log* in the processor node P2 in which T2 is running. The dependency is recorded using the instruction counts corresponding to the memory operations *Rb* and *Wb*.

When the second dependency between the processors P1 and P2 is detected due to the dependency *Ra→Wa*, FDR needs to determine, if the dependency can be transitively implied by the previous log entry or if has to be logged again in the P2's memory race log. In order to do so, P2 needs to know that the instruction count of the previous write access to the location "a" in the processor P1 is less than the instruction count that was last recorded in P2 for the processor P1 .

Thus, to implement the Netzer optimization for point-to-point logging approach, the time-stamp information (instruction count) has to be kept track of along with each cache block. The instruction count of a cache block tells the logging mechanism *when* the block was last accessed by the processor node. To keep track of this information, about 6.25% [13] of L1 and L2 cache area is required, which translates to about 128KB area overhead for a 2MB L2 cache. Further, the memory race log is buffered locally in a 32KB Memory Race Log Buffer in each processor node as described in FDR [13]. The hardware implementation of the strata logging approach is less complex, and also the strata log size is 5.8x smaller without compression and 12x with compression than that of the memory race log.

4. Using Strata to Determine Shared Memory Dependencies

In this section, we discuss an algorithm to capture the shared memory dependencies across the threads of an application. The hardware implementation of the algorithm described here will be presented later in Sections 5 and 6.

4.1 Capturing Shared Memory Dependencies using Strata

We assume a sequentially consistent memory model. In a sequentially consistent memory model there exists a total order between the memory operations executed across all the threads. All the threads' memory operations should be consistent with that total order, which means a thread's read must get the value of the most recent write in the total order. The total order must also respect a thread's program order.

Our goal is to record sufficient information during program execution, which will allow us to reproduce the total order observed during program execution while replaying.

To capture a dependency between two shared memory operations, in the point-to-point logging approach that we discussed in the Section 3, the memory count of the two dependent memory operations is logged. Instead, we propose using a logging primitive called a *stratum*. A stratum consists of the execution states in terms of the memory counts of all the running threads at the time when it is recorded. A memory count for a processor node is the number of memory operations executed since the logging began. To capture a shared memory dependency, we record a stratum just before executing the succeeding memory operation. If two memory operations are dependent on each other, we refer to the memory operation that occurred earlier in time as the preceding memory operation or simply *predecessor*. The one that occurred later in time is referred to as the *successor*. The recorded stratum separates all the memory operations across all the threads, executed before the time when the stratum was recorded from those that were executed after it was recorded. Since the stratum is recorded just before the succeeding memory operation is executed, it separates the predecessor and the successor in time.

Figure 2 shows the memory operations executed in three threads. The subscripts for the reads and writes are used to identify the memory operation. The fields inside the braces, show the address and the output value for a memory operation. The strata are represented as horizontal lines. For instance, strata $S1$ separates the successor $W2$ from the predecessor $W1$.

One advantage of using the strata to capture the shared memory dependencies is that we can apply an effective dynamic transitive optimization to reduce the size of the strata log. Also, the hardware required to implement the transitive optimization for strata is significantly less than what is required for implementing a similar optimization for the point-to-point logging approach.

We further reduce the strata log size by not logging information for WAR dependencies. This is based on our following observation: To reproduce the total order during replay it is sufficient to record strata just to capture the cross-thread RAW and WAW dependencies. We show how the cross-thread WAR dependencies can be ordered through offline analysis during replay, although minor hardware additions can be added to also log stratum for WARs. We also do not have to record strata to capture the intra-thread RAW and WAW dependencies, because those are trivially revealed by a thread's program order. Therefore, the discussion in this section focuses on capturing only the cross-thread (inter-thread) RAW and WAW dependencies using strata.

4.2 Optimizing Strata Log Size

We call the log containing the strata the *Strata Log (SL)*. We use the example shown in the Figure 2 to explain how a strata log is created. We do not have to record anything for the intra-thread dependencies. In the example, *W1→R1* is an intra-thread RAW dependency, which is revealed during replay by thread $T1$'s program order.

In a naive implementation, the SL has a stratum recorded for every cross-thread RAW and WAW dependency. However, the offline analysis algorithm only requires that, for each observed cross-thread RAW or WAW dependency, there is *at-least* one stratum in the strata log that separates the predecessor and the successor. This means that one stratum can be used to separate more than one cross-thread RAW or WAW dependency.

In the example shown in Figure 2, the stratum $S1$ is logged when the WAW dependency *W1→W2* is observed during program execution. The recorded stratum allows us to determine that the write $W1$ has to be executed before the write $W2$ during replay. Similarly, the stratum $S2$ is logged to capture the *W2→R2* dependency.

However, when the RAW dependency *W2→R4* is observed, we do not have to log a stratum. The reason is that, the stratum $S2$ is sufficient to determine that $W2$ preceded $R4$ in time. For the same reason, we do not have to log a stratum for the RAW dependency *W3→R3*. Note, $S1$ or $S2$ is sufficient to capture the RAW dependency *W3→R3*. Also note that the memory operations, $W3$ and $R3$, involved in the RAW dependency are accessing a

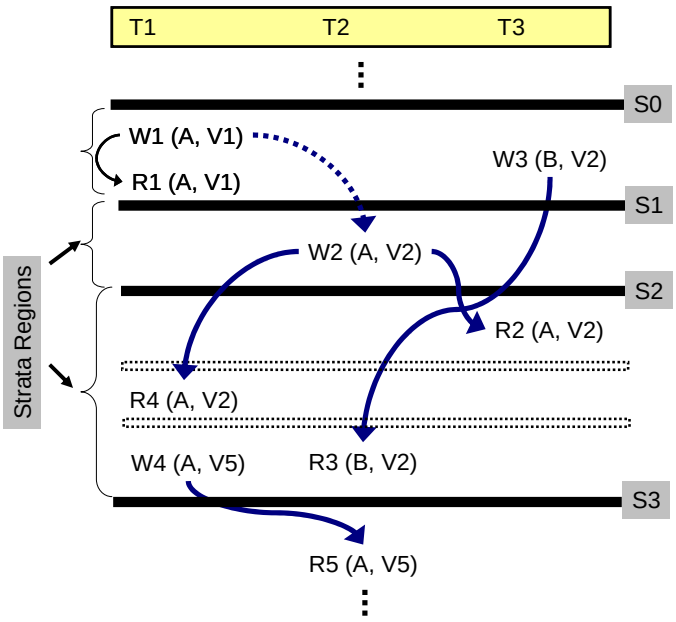

Figure 2. Recording Strata Log. The strata that are logged are shown as solid horizontal lines. The strata that are not logged by applying the transitive optimization are shown with dotted rectangular boxes. The RAW dependencies are shown as solid arrows and the WAW dependency is shown as a dotted arrow.

memory location different from the one that triggered the creation of the strata $S1$ and $S2$.

For the RAW *W4→R5* dependency, we have to log a stratum $S3$, because none of the earlier strata separate $W4$ and $R5$ in time. Thus, a stratum for a cross-thread WAW or RAW dependency is logged only if the preceding operation in the dependency gets executed after the last recorded stratum.

4.3 Advantages of Strata

We now describe the advantages of using a stratum to log cross-thread RAW and WAW dependencies.

4.3.1 Efficient Transitive Optimization

A stratum consists of the current memory counts of all the threads. In the point-to-point logging approach used in FDR [13], to record a dependency, the memory count for the time when the preceding memory operation was executed is logged along with the memory count of the succeeding memory operation. Logging the memory count for the predecessor is less "strict" than logging the current memory counts of all the threads. Therefore, a single stratum can potentially capture many RAW and WAW dependencies. Thus, the transitive optimization used to reduce the SL size is more efficient than the Netzer transitive optimization used for point-to-point logging [13, 6].

4.3.2 No WAR Logging

Capturing shared memory dependencies using strata allows us to ignore WAR dependencies while logging and determine them offline. Not capturing WAR, reduces the strata log size and the hardware support required for recording them using strata.

4.3.3 Efficient Hardware Implementation

The amount of hardware required to implement the transitive optimization to reduce the strata log size is significantly less than what is required to implement the Netzer optimization in FDR [13].

4.3.4 Supports Snoop Based Systems

In a directory system, when a dirty block is evicted, which processor or thread last accessed the block could be maintained in the directory. However, in a snoop based system that information is lost, which is a problem for the point-to-point logging approach used in the prior works [13, 6]. This problem can be solved if we use the strata for logging RAW and WAW dependencies, because we do not require precise information about the predecessors. We just need to detect if there is a RAW or WAW dependency due to the evicted cache blocks. We detect that by keeping track of the set of evicted blocks in a bloom filter (a bit vector indexed by the hash of an address). The hardware implementation for a snoop based system will be described in detail in Section 5.

4.3.5 Handles Paging

In Section 3, we pointed out a corner case in dealing with OS paging in the middle of recording a program's execution using the point-to-point solution. When a page is swapped out, all the information about who accessed what block and when it was accessed is lost, as it can be found neither in the private caches of the processor nodes nor in the directory.

To address this problem, we log one additional stratum when a page is re-mapped. The recorded stratum separates the memory operations executed before and after the paging activity, which is sufficient for the offline analysis to determine the dependencies. Since the paging activity is less frequent, when compared to the frequency at which we create strata during normal execution, additional strata logged due to paging constitute a small portion of the strata log.

4.4 Off-line Analysis to Determine a Total Order

During replay, assume for now that we know all the memory operations executed in each thread along with their addresses in the program order. We postpone the discussion on how to get this information during replay for the BugNet and the FDR approaches to Sections 4.4.1 and 4.4.2 respectively.

Given the above information, using the strata log, we can infer the total order between the memory operations observed during logging. We use the example shown in Figure 2 to explain our algorithm. Figure 2 shows the memory operations along with their addresses and output values, but for our offline analysis we do not require information about the output values.

We define a *strata region* to consist of all the memory operations executed across all of the threads between two strata. Figure 2 has three strata regions: $S0-S1$, $S1-S2$ and $S2-S3$. There is a total ordering between the strata regions because the strata are ordered by time. Therefore, if we order the memory operations executed in each strata region in isolation, and then order all the memory operations across strata, we will get a total order for all the memory operations. We first discuss how to order the memory operations executed within a strata region.

The memory operations executed in a thread are ordered by the program order. For the example shown in Figure 2, we know that $W4$ was executed after $R4$. Hence, we have to determine only the cross-thread dependencies. However, within a strata region, there cannot be any cross-thread RAW or WAW dependency. This is true because the strata log is created in such a way that there is at-least one stratum that separates the memory operations involved in a cross-thread RAW or WAW.

The above property simplifies our job to finding the cross-thread WAR dependencies within a strata region. Since we are assured that there is no cross-thread RAW dependency within a strata region, if we find a read and a write in two different threads, such that the addresses for both the operations are the same, then we are guaranteed that the write has to be executed after the read during

replay (WAR dependency). For example, in Figure 2, both $R2$ and $W4$ access the same memory location and they are within the same strata region $S2-S3$. If $R2$ was executed after $W4$ during logging, a stratum would have been logged between the two operations. Since there is no stratum separating the two, we know that the dependency between those two operations is a WAR dependency, *R2→W4*.

Once we have identified all the cross-thread WAR dependencies for a strata region, we can determine a valid total order for the memory operations of a strata region. While determining the total order, we make sure that the program order is preserved in addition to the inferred WAR dependencies. For example, a valid total order for the memory operations of the strata region $S2 - S3$ is the following: *R2→R3→R4→W4*. For the strata regions $S0 - S1$ and $S1 - S2$ there are no cross-thread WAR dependencies. Hence, for those regions we just need to make sure that the program order is preserved. A valid total order for the strata region $S0 - S1$ is *W1→R1→W3*.

Now that we have determined a total order for the memory operations of each strata region, we can order all the memory operations using the recorded total order for the strata regions. For example, we know that $S0 - S1$ happened before $S1 - S2$, $S1 - S2$ happened before $S2 - S3$ and so on. Therefore, the memory operations of the strata region $S0 - S1$ should precede the memory operations of the strata region $S1 - S2$ in the total order. We can therefore determine a total order for all the memory operations within these three strata regions. In our example, a valid total order is *W1→R1→W3→W2 →R2→R3→R4→W4*. However, we obtained this total order based on the assumption that we have knowledge of all the memory operations and the addresses that they accessed. The next two sections explain how to obtain this information during replay using the checkpoint logs of BugNet [6] and FDR [13].

4.4.1 Load-based Checkpoint

The BugNet's First Load Log (FLL) [6] is created for each thread. It captures the values of the load instructions executed in a thread, which is sufficient to deterministically replay that thread. It is sufficient even in the presence of shared memory updates, because a load accessing an address written by another thread will notice that the address' value has changed, and will log that value in the its FLL.

By deterministically replaying each thread individually, we create a *replay* trace for each thread. In the *replay trace* of a thread, we have information about all the memory operations executed by that thread in the program order along with the addresses that they accessed. Using this information, and the offline analysis described earlier, we can derive a total order between all the memory operations.

4.4.2 Copy-On-Write Checkpoint

FDR [13] uses a copy-on-write checkpoint scheme along with a *redo* log to deterministically replay the full system. In a copy-on-write checkpoint scheme, whenever a memory location is updated, the old value residing in the memory location is logged. In addition, the final state of all the memory locations is logged at the end of a checkpoint. Using the final state, and the log of memory updates, one can determine the memory values at the beginning of a checkpoint. With this information, one can start replaying. However, during replay, we also have to reproduce all the system interactions and the shared memory dependencies. To reproduce the system interactions (like interrupts, system calls and DMA transfers) FDR explicitly logs such information in what we call a *redo* log.

In order to deterministically replay using the FDR checkpoint logs, we need information about the shared memory dependencies. The strata log that we described earlier can be used for this purpose. We can deterministically replay the full system using the FDR's copy-on-write logs and the strata log as follows.

We start the replay from the first strata region in the checkpoint and then proceed to replay the following strata regions in order. However, it is not straightforward to replay from the start to the end of a strata region without the knowledge of potential WAR dependencies that may exist within the strata region. We solve this problem by performing a search through the possible memory orderings for a strata region.

We first begin the replay for a strata region without assuming any WAR dependency and the only order we preserve is the program order. During the search, we may observe a read and a write executed in two threads with the same address. We know for sure that this is a WAR dependency and not a RAW dependency, because during logging, we create strata in such a way that there are no RAW or WAW dependencies within a strata region. However, in our replay experiment, while searching for a correct memory ordering for the strata region, we might have executed the write before the read. If so, we take note of the WAR dependency and start replaying again from the start. In the subsequent replay experiments to find a correct memory ordering, we enforce the WAR dependences that were found in the earlier replay experiments.

During replay, we are guaranteed to not wander down a control path that is different from the recorded program execution. For that to happen, some load would have to have *read* an incorrect value written by another thread during the replay. However, that would be a cross-thread RAW dependency, which is not valid, since there cannot be any RAW dependencies within a strata region. During replay, if we find a RAW dependency, then this means that this is really a WAR dependency, and we take note of this newly found dependency and restart replay from the beginning of the strata region. In that replay and the subsequent replays, we will not allow the write to execute till the dependent predecessor read in the other thread has executed. For example, consider the strata region $S2-S3$ in Figure 2. During our replay search for a correct memory order, it is possible that the write $W4$ is executed before the read $R2$. After noting this WAR dependency, in our subsequent replays, if we reach $W4$ before executing $R2$, we will stall the thread T1 till $R2$ in thread T3 has executed.

We continue the above process till we are able to replay up to the next stratum without encountering a RAW dependency. This gives us a final memory ordering for the strata region, and that is used for deterministic replay debugging. This ordering lists instructions in the same order as observed during program execution.

4.5 Correlating Strata Logs to BugNet/FDR Checkpoint Logs for Replay Debugging

Each processor node keeps track of the memory count, which is the number of memory operations that it has executed. This is the running count of memory operations executed on that processor, since logging began. We assume a 32-bit counter for the results in this paper. When a new checkpoint is created in FDR [13] or BugNet [6], the current memory count value is stored in the new checkpoint header. In the case of BugNet, a per thread first load log (FLL) is created as part of the new checkpoint. We store in the checkpoint header of the FFL the memory count of the processor that the log is being generated on. In the case of FDR, a global checkpoint is created. The checkpoint header of the global checkpoint contains the memory count of all the processor nodes. By logging the memory counts in the checkpoint headers, we know exactly where the checkpoint logs fit into the time line of the strata logs.

The strata are logged based on the memory count values tracked in each processor node. Thus, it is easier to map the strata logs to the checkpoints that they correspond to. However, to reduce the strata log size we examine a compression technique that logs the difference between memory counts in 16-bit values. Instead of logging the 32-bit memory count value, we just log the difference between the memory count for the processor in the immediately preceding stratum and the current memory count for that processor. Even when we log just the stride values, it is still possible to correlate the strata logs with the checkpoint logs of FDR and BugNet, because at the beginning of the strata log we log the full memory count values of all the processor nodes. If we start a new strata log, the new strata log is also created with the current memory count of all the processors in the log's header. This allows us to correctly map the strata log entries with the FDR and BugNet checkpoint log headers.

With the above information we can replay the program's execution for deterministic replay debugging using the BugNet/FDR checkpoint logs in combination with the strata logs, as long as we have a little more information about system events. For BugNet [6], the only other piece of information needed for deterministic replay is the order of thread context switching. To address this, BugNet has a context switch log to record the time of the context switch using the memory count of the processor, as well which thread is context switched in and which thread is context switched out for the processor. For FDR [13], it does not need a context switch log, since it can deterministically replay the operating system thread scheduling, but it does need the redo log, which provides the ability to replay all of the inputs to the system.

Since these systems provide deterministic replay, and we now have a total order for the memory operations, they can be used to single step through multi-threaded execution for debugging. This allows the developer to observe the interaction between the threads through the shared memory reads and writes, which is useful to track down bugs due to data races.

4.6 Processor Effects on the Logging

We now discuss how to handle logging at the block level, prefetching, and how out-of-order execution affects the strata logs.

4.6.1 Capturing Dependencies at the Cache Block Level

We detect the shared memory dependencies at the granularity of cache blocks. This is because, we detect dependencies by observing the cache coherence messages, which operate at the granularity of the cache blocks. As a result, we might detect a false shared memory dependency (due to two processors accessing different words in the same cache block), and log a stratum for it.

However, the above is not an issue for our offline analysis. When a false dependency is detected, in the worst case, one additional stratum is logged. This is not issue, because the additional stratum just specifies a much stricter (but still a valid) ordering between memory operations.

4.6.2 Prefetching and Out-of-Order Execution

A hardware prefetcher or a software prefetch instruction can bring a memory block into the cache which might not be eventually used (read or written) by the processor. This might result in unnecessary strata being logged. However, additional strata do not compromise correctness.

Non-blocking caches and out-of-order execution in modern processors can send or receive a coherence request/reply for a cache block out-of-order (out of program order). In systems implementing aggressive speculation, a cache block may be accessed even before its coherence is done. However, even in these systems, if the processor supports sequential consistency (which is what we assume and model), then it makes sure that the cache access and the coherence activity appears to be in the commit order of the instructions (program order). Therefore, our strata logs and coherence messages associated with the strata logs are consistent with the program commit order.

4.7 Future Work

We just showed how to determine the cross-thread WAR dependencies with the help of offline analysis and the strata log. It could be possible to determine even the WAW dependencies in the same way. Assume we only log strata for RAW dependencies. If we do not log any stratum to capture the WAW dependencies, then a strata region can contain a WAW dependency. Therefore, we would have to order the writes involved in a cross-thread WAW dependency during offline analysis. This analysis is more complicated, since when we see a WAW dependency within a strata region we cannot deterministically order the two writes, unless we know exactly what the strata region's live-out value for that address is and we know what the value of the two writes are.

In Section 4.4.1 we explained how using BugNet's [6] First Load Log for a thread one can deterministically replay that thread. Thus, we can know not only the memory operations and their addresses, but also their values. Figure 2 shows the values of the memory operations in the second field. It is possible to obtain all these values if we are replaying using the BugNet's FLL. Therefore, it is possible to determine the live-in of each strata region using these logs, as well as the live-out. Using this information for a strata region, we can determine the last writer to an address in the strata region, and can then deduce the write-after-write's off-line. However, this offline algorithm is much more complicated and the total order that we find during offline analysis may not be exactly the same as what was observed during logging. We plan to explore this logging optimization and its implications in the future.

5. Hardware Implementation for Snoop-based Systems

In this section, we discuss how we support creating Strata Logs (SLs) for snoop-based systems. As we discussed in Section 3, the previous Point-to-Point approaches [6, 13] cannot be easily implemented in a snoop-based system.

5.1 Detecting Cross-Node RAW and WAW for Cached Blocks

To explain our approach, let us assume for now that the caches are of infinite size. This means, once a processor node accesses a memory block, it stays in its private cache till another processor writes to it. If another processor writes to the block, then it gets invalidated. Therefore, a memory block once fetched into some processor's cache resides in at least one of the processors' caches throughout the lifetime.

Our goal is to detect cross-node RAW and WAW dependencies. We achieve this by monitoring the coherence messages. Whenever a processor node encounters a read or a write miss for a memory block, it places a request on the bus. If any other processor node has a dirty copy of the memory block, which means the processor wrote to the block, then there exists a RAW or a WAW dependency. Therefore, when the owner of the block replies on the bus, the reply is piggybacked with a *log stratum* bit, whose value is set. The log stratum bit instructs other processor nodes in the system to log a stratum. Each processor node, logs their current memory count in their strata log. Our design ensures that the memory count logged for the processor that generated the read or write miss, corresponds to the memory operation executed prior to the read or write. This ensures that the stratum separates that read or write from all the prior memory operations.

Note that the memory count for each processor representing the stratum is logged into each processor's own strata log. Therefore, we need to be able to construct a global strata log from the individual per processor strata logs. However, we create the strata logs in all the processor nodes at the same time. They are initialized with the full 32-bit memory count values of the respective processor nodes at the time of creation. Thus, the strata logs across all the processor nodes always stay synchronized. That is, the first entry in a processor's strata log corresponds to the first entry in every other processor's strata log, and it is the same case for the rest of the entries in the strata logs as well.

5.2 Detecting Cross-Thread RAW and WAW for Evicted Blocks

The previous section assumed infinite caches. Let us remove this assumption. With finite size caches, there exists an issue for the snoop-based protocol when a dirty block is evicted from the cache. When a memory block is evicted out of the cache, information about the last writer to that block is lost.

We solve this problem for snoop-based systems using a separate bloom filter [10] in each processor node. Our bloom filter is a bit vector indexed by a hash of the memory address. Since we are interested in detecting only the cross-node RAW and WAW dependencies, we must keep track of the fact that there was a writer to this memory block. Hence, whenever a dirty block is written back (evicted) to main memory over the bus, all the other processor nodes snoop the bus, and set the bit in their private bloom filters by indexing them using the hash of the physical address of the memory block that is being written back.

If a processor node encounters a read or write miss for a memory block, it checks its private bloom filter to see if, in the past, some other processor node had written to that memory block. If the bit for the block is set in the bloom filter of the processor that encountered the read miss, then we know that there may be a potential cross-node RAW or WAW dependency. Hence, a stratum has to be logged. To log the stratum, we piggyback the coherence request message (generated by the processor that encountered the read/write miss) with the log stratum bit set to true. All the processors snooping the bus will see a set stratum bit. This forces each processor node to log its current memory count in its strata log. Whenever a new stratum is logged, all the processors clear all of the bits in their bloom filters. We can clear all of the bloom filters, because the recorded stratum separates all the reads and writes that follow the stratum from those writes that were executed in the past. Essentially, by clearing the bloom filters we are implementing the transitive optimization for the writes to the uncached blocks.

The bloom filter essentially predicts whether an uncached block was written after the last recorded stratum. The bloom filter guarantees that there are no false negatives. That is, we will not miss any RAW or WAW dependency due to aliasing in the hash indexed bit vector. However, there could be false positives, which means that we may end up logging a few more strata than we need to. For our results, we use a bloom filter of size 128 bytes (1024 entries, one bit per entry) per processor-node, which resulted in less than 1% of additional strata for most programs we examined.

5.3 Implementing Transitive Optimization for Cached Blocks

When we detect a cross-node RAW or WAW dependency, we do not have to log a stratum if the write operation involved in the dependency was executed before the last recorded stratum. Earlier, we explained how this optimization is implemented for uncached blocks by just clearing the bloom filter bits when a stratum is logged. If the dirty block is cached in one of the processor nodes, then we need a way to know if the write to that block occurred before or after the last recorded stratum.

Unlike the Netzer transitive optimization used in the previous Point-to-Point proposals [13, 6], which required storing the instruction count with each cache block, in our approach all we do is associate a single bit with each cache block. We call this bit the *dependence bit*. The dependence bit for a cache block indicates whether the cache block was written before or after the last recorded stratum. The dependence bit for a cache block is set whenever there is a write to the cache block and is reset whenever a stratum is logged.

When we have a read or write miss, and a dirty block is found in another processor's cache, then this means that there is a RAW or WAR dependency, but we *log a stratum only if the dependence bit is set for that cache block*. Also, while evicting a dirty cache block, *the bloom filters of the processor nodes are updated only if the dependence bit for the evicted cache block is set*. If the dependence bit was not set for the dirty block, we do not need to keep track of it in the bloom filter nor log a stratum. This is because a stratum has already been logged since the last time the block was modified.

Whenever we log a stratum, in addition to clearing the bloom filters for all the processor nodes, we also clear all the dependence bits in all the caches. This is valid, because all the writes that were executed before logging the stratum are separated from the memory operations that are going to be executed after recording the stratum.

5.4 Recording Stratum for WAR

We need to capture just the RAW and WAW dependencies like we described in Section 4.4. However, in our experimental evaluation to be discussed in Section 7, we studied the size of the strata log required to capture *all* the shared memory dependencies, including the WAR dependencies. We briefly discuss here, the additional hardware required to capture the WAR dependencies.

First, we need to be able to detect the WAR dependencies. This is straight-forward as long as the blocks are cached. However, if a block read by a processor is evicted, we need to keep track of it similar to how we tracked the dirty block evictions. This requires a broadcast on the bus even when a clean cache block is evicted, which is not normally required in a MESI protocol. Each processor node snoops the broadcast message to set the corresponding bit in the private *read* bloom filter. Note that this bloom filter is an addition to the *write* bloom filter already used for tracking the evicted dirty blocks. Also, we need an additional dependence bit that tells us whether the block was read by the processor before or after the last recorded stratum.

Because of this additional hardware complexity, our primary solution focuses on recording strata for only RAW and WAW, and determining RAW using offline analysis as described in Section 4.4.

5.5 Hardware Comparison to Point-To-Point Logging

The additional logic added to our architecture are the bloom filter per processor, and one dependence bit per cache block in the private caches of each processor. In our scheme, we do not have to tag each memory block with the instruction count like in the prior works [13, 6] to record the shared memory dependencies. We therefore avoid the additional 6.25% area overhead in the L1 and L2 caches used in the prior techniques.

6. Hardware Implementation for Directory Based Systems

In this section, we discuss how we can capture the shared memory dependencies using the Strata Log for directory based systems. Figure 3 shows the changes required in a directory based system to record the strata log. It shows one processor node in the multiprocessor system. It also shows the directory controller where we record the strata log. For our simulations we use an 8 KB hardware buffer to buffer the writing of the stratum to the strata log in main

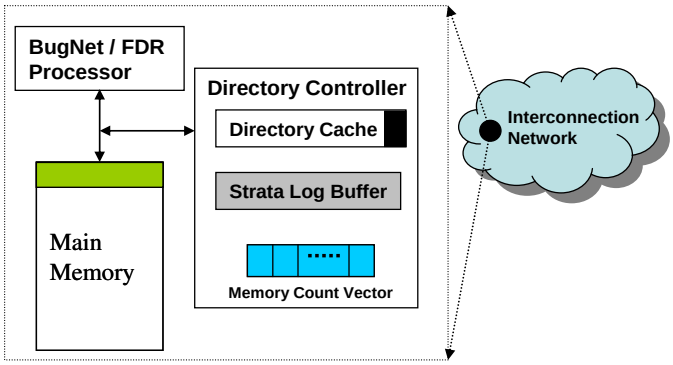

Figure 3. Strata logging support in a directory based system.

memory. We also store a memory count vector in the directory as well as dependence bits in the directory and per processor cache, which we will now describe in this section.

6.1 Capturing RAW/WAW using the Strata Log

To explain our approach, let us assume a system with a centralized directory, and that the directory knows for each block if it has been written since the last stratum was logged. We will remove these assumptions later in this section. Since the processor nodes are not connected by a common bus like in the snoop-based systems, it is not efficient to create the strata logs in the processor nodes. We instead chose to create the strata log in the directory controller.

When a cross-node RAW or WAW dependency is observed at the directory, a stratum has to be logged by the directory controller. To log a stratum, the directory controller needs to know the current execution states (memory counts) of all the processor nodes. One way to achieve this is by polling all the processor nodes for their current memory counts. However, this incurs a heavy communication cost. To avoid this communication cost, we instead propose to have each directory controller keep track of a vector of memory counts (MVector), one for each processor node in the system. Each count represents the last time the processor accessed the directory. A stratum is logged using these memory counts. The vector of memory counts is updated as follows. Every time a processor node performs a read or a write request to the directory due to a cache miss, it piggybacks its current memory count along with the coherence request. Also, when a dirty block is written back to memory, the current memory count is also piggybacked in the write update coherence message. We update the MVector for the processor each time the memory count is piggybacked on a coherence message.

In this scheme, some of the memory counts in the vector can be stale (not up-to-date) when a stratum is logged, relative to the current memory counts on all of the processors. This is fine, since the memory counts that the directory sees can be used to log a stratum across all of the processors, which is valid in terms of capturing the shared memory dependencies. Consider the example shown in Figure 4. The example shows four processor nodes. The read and write operations are shown along with their addresses. The RAW and WAW dependencies are also shown using arrows. In the figure, assume that the stratum $S0$ has been logged due to some previous dependency. The processor nodes $P3$ and $P4$ update their memory counts in the directory controller when they encounter read misses (due to $R1$ and $R2$) for the address C. Later, when $P1$ sends a write miss request for $W2$, a WAW dependency is detected with $P2$, which currently has a dirty copy of the memory block (written by $W1$). A stratum is logged to capture this *W1→W2* WAW dependency. The directory controller logs the stratum $S1$ using the memory counts in its MVector. Note that the memory counts for the processor nodes involved in the dependency are always up-to-date when the stratum for the dependency is logged. In our example, the memory counts of $P1$ and $P2$ are up-to-date when the stratum $S1$ is logged. Since the stratum will separate the dependent operations in time, the memory count logged for the processor node $P1$ is one less than the memory count corresponding to the write operation $W2$.

After logging the stratum $S1$, the processor node $P2$ encounters a write miss for $W5$ and updates the memory count in the MVector. Later, the RAW dependency *W3→R4* is observed between $P4$ and $P3$, when $P3$ encounters a read miss for $R4$. The directory controller logs the stratum $S2$ to capture this RAW dependency. Note that the memory counts are up-to-date for $P3$ and $P4$ while logging the stratum $S2$. However, it is not the same case for $P1$ and $P2$. In fact, for $P1$, the memory count logged in $S2$ is same as the memory count that was updated when $P1$ sent a coherence request for the write $W2$.

In spite of using stale memory counts while logging the strata, the following two properties that are essential for our offline analysis are still preserved: (1) There exists at least one stratum between the memory operations involved in a cross-node RAW or WAW dependency. Thus, a strata region cannot have any cross-node RAW or WAW dependencies. (2) The strata regions are non-overlapping, because the value of a memory count in the MVector either increases or stays the same. These properties allows us to consider one strata region at a time during offline analysis, and determine a total order for the memory operations executed with a strata region.

6.2 Determining RAW and WAW Dependencies in the Directory

Similar to our snoop-based implementation, we use a dependence bit per cache block to determine if a dependency needs to be logged. We also have a dependence bit for each directory entry in the directory cache. The dependence bit in the processor's cache is set whenever the processor writes to the cache block. In the directory systems, the dependence bit in the directory cache is set whenever a dirty block is written back to the memory and that dirty cache block has its dependence bit set.

While servicing a processor's read or write miss request for a cache block, a potential RAW or WAW can exist if some processor node in the system has exclusive access to the cache block. In that case, the directory controller sends a data fetch request to the processor node that has exclusive access. The processor node with exclusive access to the cache block, piggybacks the value of that block's dependence bit on the coherence reply message to the directory. We detect a RAW or WAW dependency only if the dependence bit information received through the coherence reply is set to true (that is, the owner had written to the cache block).

In case when the block in not cached in any of the processor node, while servicing a processor's read or write miss request for a cache block, we detect a RAW or WAW dependency if the dependence bit is set for the directory entry of the cache block. If the bit is set, we log a stratum to separate the dependency. If the directory cache cannot keep track of every block in memory, and we get a miss in the directory cache for the memory block, then we conservatively log a stratum on a directory cache miss. Thus, the dependence bits in the private caches of the processor nodes and in the directory cache allows us to detect the RAW and WAW dependencies and log a stratum appropriately to capture them.

When a directory logs a stratum, the dependence bits for all the entries in that directory can be cleared. This is because, any write that had set a dependence bit in the directory entry earlier is ordered by the stratum that the directory is currently logging. To reduce the

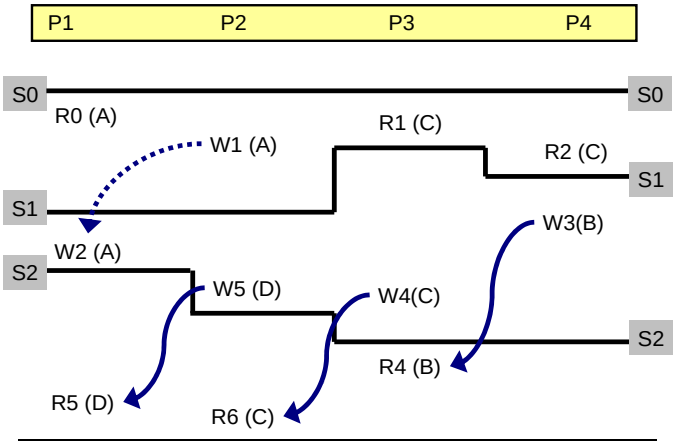

Figure 4. Example of a Strata log in a directory based system.

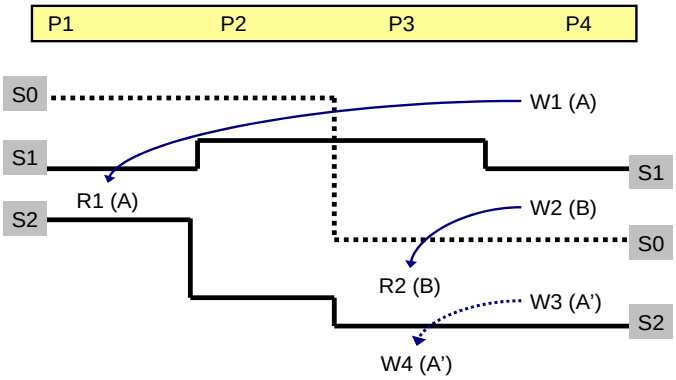

Figure 5. Strata log collected in a system with two directories.

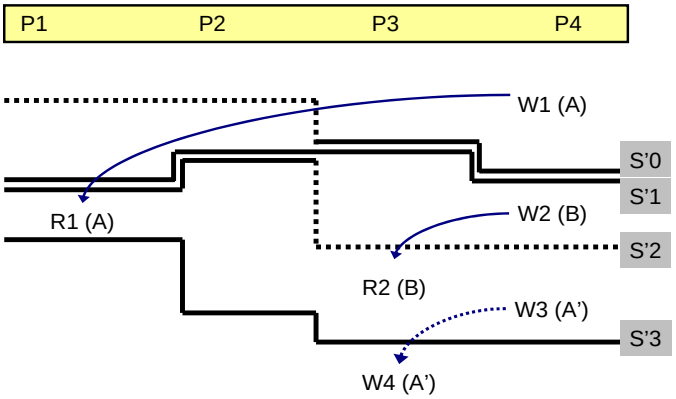

Figure 6. Combined strata log created from the strata logs of two different directories during replay.

amount of logging as much as possible, we would also like to clear the dependence bits in the processor caches, whenever a stratum is logged. We would only be able to do that for all of the processors, if the stratum contained the current memory counts for all the processor nodes. However, this would require additional coherence messages between the directory and every other processor node in the system. Therefore, we chose to reset the dependence bits only for the two processor nodes that are involved in the RAW/WAW dependency that triggered the creation of the stratum. This is valid because the new stratum contains the up-to-date memory counts for those two dependent processor nodes. Hence, all the writes executed in those two processors before logging the stratum are ordered by the recorded stratum.

Consider again the example shown in Figure 4. Assume that after the write $W5$, the dirty block (with address D) was immediately written back to the memory. This sets the dependence bit in the directory entry corresponding to the memory block with address D. When the stratum $S2$ is logged, the dependence bits in the directory are reset. This includes the dependence bit for the block D. Later, when $R5$ accesses the same block, a new stratum is not logged, even though there is a cross-node RAW dependency (*W5→R5*), because the stratum $S2$ already separates $W5$ and $R5$.

Now let us explain another example to show how the dependence bits in the caches are used. The processor node $P3$ executes the write $W4$ to the address C and sets the dependence bit in its cache block. The cache block remains in the dirty state until the time when processor node $P2$ executes the read $R6$. Clearly, there is a cross-node RAW dependency *W4→R6* between $P3$ and $P2$. However, when the stratum $S2$ was created for the RAW dependency *W3→R4*, the dependence bits in the private caches of $P3$ and $P4$ would have been reset. As a result, the dependency information piggybacked along with the coherency reply from $P3$, to service the read miss for $R6$ *W4→R6*, tells the directory that a new stratum to capture the RAW dependency *W4→R6* is not required. Thus, a stratum is not logged for the RAW dependency *W4→R6*.

6.3 Strata for a Distributed Directory

In this section, we relax the assumption of a centralized directory and show that our approach is applicable for distributed directories as well. In the case of distributed directories, each directory controller captures the dependencies for the addresses that it services using a strata log. So we have multiple strata logs, which are combined into one unified strata log during offline analysis. Figure 5 shows an example of two strata logs, collected in a distributed directory system with two directories. The solid strata are collected in one directory and the dotted ones are the strata collected for the other directory.

A strata log collected in a directory serves the purpose of determining the dependencies between the memory operations accessing the addresses mapped to that directory. Therefore, we are still guaranteed that each cross-node RAW and WAW dependency is separated by at least one stratum in one of the strata logs. In Figure 5, addresses A and A' are assumed to be mapped to one directory (the strata log for this directory is shown using solid lines). The address B is mapped to the other directory. It can be seen that the solid strata $S1$ and $S2$ separate *W1→R1* and *W3→W4* dependencies respectively, while the dotted stratum $S0$ captures the *W2→R2* dependency.

However, unlike in a centralized directory, the strata regions in the strata logs collected in different directories can be overlapping. The reason for this is that the MVectors used to log the strata across the directories are not updated in the same way. An entry for a processor node in the MVector is updated only when that processor node communicates with that directory to resolve a miss or when it is writing back a dirty cache block.

Overlapping strata regions are an issue, because in the offline analysis that we described in Section 4.4, one strata region is analyzed at a time. To solve this problem, in our offline analysis, we first combine the multiple strata logs for the different directories such that there are no overlapping regions in the combined strata log. When combining the strata logs, we look to see if there are any two strata that are intersecting. A stratum in each log contains the memory counts for each processor when the stratum was recorded in the directory, and from these counts we can easily determine if two strata are intersecting. If there are two intersecting strata, we use their memory counts to make non-overlapping equivalent strata, which are put into the combined strata log. For example, in Figure 5 the strata $S0$ and $S1$ are intersecting. For these two strata, we create three strata $S'0$, $S'1$ and $S'2$ in the combined log in such a way that none of the new strata intersect, and these new strata still separate the regions of memory operations that the two strata were originally created to separate. Figure 6 shows the combined strata log. It is stricter because the number of strata, between any two dependent memory operations, is either the same as before removing the intersections or greater. For example, in Figure 6, the reads and writes involved in a RAW or a WAW dependency are still separated by at least one stratum. There are two strata $S'0$ and $S'1$ that separate the cross-node RAW dependency, *W1→R1*, whereas in the strata log shown in Figure 5 there is only one stratum $S1$ to separate those dependent operations. The RAW dependency *W2→R2* (observed in the second directory) is captured by the stratum $S'2$.

Once we have the strata log with non-overlapping strata regions, we can use the offline analysis that we described in Section 4.4 to determine a total order for all the memory operations.

6.4 Recording All Shared Memory Dependencies

If we wanted to also capture WAR shared memory dependencies with our strata approach, we need to have read dependence bits for reading a block in the processor's caches and in the directory cache. These additional dependence bits are required to determine whether there was a read to a memory block after the last recorded stratum. When there is a write miss, we can detect if there is a WAR dependency, and using the read dependence bit information we can decide whether to log a stratum or not.

6.5 Hardware Requirements

The additional hardware states required for creating the strata log is just one dependence bit per cache block in each processor node and one dependence bit per directory entry in the directory.

7. Results

To create the logs, in this study, we used Simics [5] to capture the logs as well as to model the architecture support required for logging. In Simics, we modeled a four node CMP processor with 64KB L1 caches and a 2MB L2 cache, and modeled both directory and snoop protocols. We also built a replayer in Pin [4] that consumes the logs to provide deterministic replay debugging of the programs.

To evaluate our shared memory dependency logging improvements, we used data parallel programs from the Splash benchmark suite [12]. We could only get five of the main benchmarks to compile, and we provide results for all of these, which are `barnes`, `ocean`, `radiosity`, `raytrace` and `water`. We focus on these for tracking shared memory dependencies, since they represent a workload which will stress the shared memory dependency logging. We ran each program configured with 5 threads on a 4 processor node system. Each program was run until the total number of memory operations across all the threads reached 400 million memory operations per program. We found this to roughly execute about 100 million memory operations per thread.

7.1 Logging Performance Overhead

In terms of performance overhead, we found that the logging incurs only a 1% slowdown due to the extra memory traffic from logging, which is consistent to the low (few percent) overhead reported in FDR [13] and BugNet [6]. This is because the logs are non-cachable writes to memory, so they do not pollute the caches at all, and they have low priority for the bus access.

7.2 Strata Logging Results

In Figure 7, we present results for the number of log entries to capture shared memory dependencies. The x-axis represents the programs and the y-axis represents the number of log entries generated by the point-to-point (P2P) scheme with Netzer optimization, our new Strata directory and Strata snoop architectures. Results are shown in terms of the average number of log entries per 1 million memory operations. The figure shows five bars. We first concentrate on the first, second and fourth bars, representing P2P, Strata for Directory and Snoop cache coherence protocols. A log entry for P2P is created for every RAW, WAW and WAR shared memory dependency. The log entry for our Strata results represent the number of logged strata, when logging strata for only RAW and WAW. The results show that Strata has significantly less log entries when compared to P2P. For our directory approach we require 10.5x less number of logs than P2P, and for snoop based system the number of log entries is reduced by a factor of 9.9x.

The reasons for these savings are two-fold. First, Strata only records RAW and WAW dependencies, therefore saving all log entries related to WAR dependencies. More importantly, our scheme implements a transitive optimization, which yields significant savings in reducing the number of stratum to be logged. Strata Snoop does slightly worse than Strata Directory because of aliasing in the bloom filters.

In Figure 7, the third and fifth bars (All) show the number of stratum log entries if we create a stratum for logging all shared dependencies (including WAR), as described at the end of the prior two sections. Adding WAR stratum logging incurs an overhead of 25% additional log entries when compared to logging only RAW and WAW strata. In addition, it requires the extra hardware explained in Sections 5 and 6, in order to monitor whether reads occurred after the last logged stratum. From the results, we see that most of the savings, in terms of the number of log entries come from our stricter (and more efficient) stratum transitive optimization, which results in significantly fewer log entries compared to P2P, which applies point-to-point Netzer optimization.

Figure 8 shows the log sizes in terms of bytes for every 1 million memory operations (y-axis) for each approach. The first bar shows the P2P approach, which store 9 bytes for each shared memory dependency logged as described in FDR [13]. The second and the fourth bars show the log sizes for our Strata Directory and Strata Snoop approaches logging strata for only RAW and WAW. The third and the fifth bars show the log sizes for logging stratum for RAW, WAW, and WAR. For every stratum four words are logged, one word is required for the memory count of each processor. These are our uncompressed results. When compared to P2P, our shared memory dependency logs are 6x and 5.6x smaller on average for the directory and snoop-based systems respectively, when logging only the RAW and WAW dependencies. If all the dependencies are logged, the ratios go down to 4.5x and 4.4x for the directory and snoop-based systems respectively. Figure 8 shows a 25% reduction in log size for not having to log WAR dependencies with Strata. The biggest advantage of only logging WAW and RAW stratum

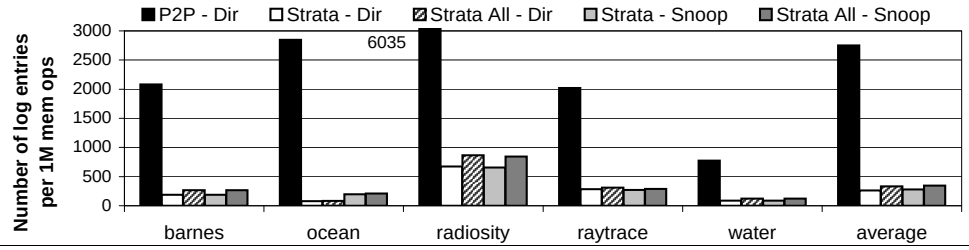

Figure 7. Number of P2P and Strata Log Entries

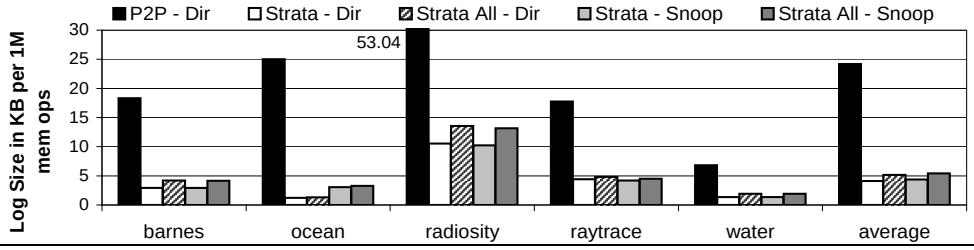

Figure 8. Log size for Recording Memory Dependencies

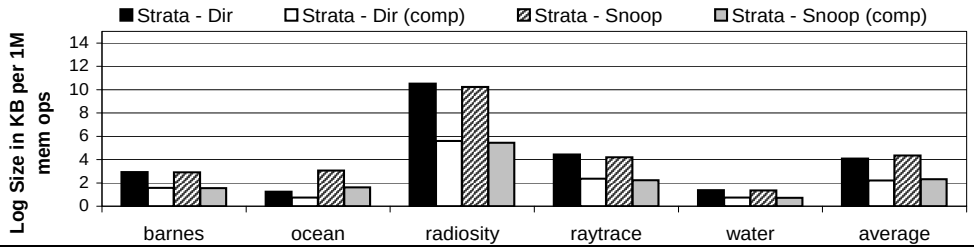

Figure 9. Compressed Log sizes for Recording Memory Dependencies

and figuring out the WAR off-line, is the avoidance of the additional complexity to log the WAR stratum described in the prior sections.

Figure 9 shows the results for Strata log sizes without and with compression. We show results for logging only the RAW and WAW dependencies, since the same trend holds for logging all dependencies as well. Without compression, the Strata approach logs 4 words for each stratum logged, as described before. For the compressed results, instead of logging a full 32-bit memory operation count for each processor, we log only 16-bits if the memory count stride (difference between the previous stratum memory count and the current stratum memory count for a processor) can be expressed using 16-bits. In addition, we need one bit field for each memory count entry per processor to distinguish between the two formats. Therefore, we approximately have two words per log entry after performing this compression. The first two bars show the results for the Strata Directory approach without and with compression, and the last two for the Strata Snoop approach without and with compression. The results show that with this simple form of compression, our log sizes are 47% smaller than not using the compression. When using compression, the Strata log sizes are on average 12x times smaller than P2P.

Overall, the results show that the storage overhead of logging the shared memory dependencies for P2P is 24KB for 1 million memory operations, and for our approach it is 4.1KB for directory and 4.4KB for snoop-based protocols for every 1 million memory operations without compression. With compression, the sizes are 2.2KB and 2.3KB for the directory and snoop-based systems. To put these log sizes in perspective with the rest of the logging done for BugNet, for these programs, the First Load Log (FLL) size required to capture the execution for 1 million memory operations is 26.6 KB on average without compression. Therefore, the Strata logs account for about 15% of the total log storage needed to provide deterministic replay with BugNet for these programs.

Another interesting observation is that the Snoop approach results in more log entries than the Directory approach. This is especially visible for the program `ocean`, as seen in Figure 7. This is caused by aliasing in the bloom filters, which result in false positives when detecting dependencies with uncached blocks. We set the bloom filter when a dirty copy of a block is written back to memory. However, if another entry aliases to the same bloom filter entry, then on a miss, we would detect a dependency between two operations, which in fact are not dependent, and log a stratum. Note that this is not a issue. It just results in redundant strata log entries, and larger log sizes. To measure the effects of aliasing in our schemes, we measured how many strata were logged due to aliasing. For all benchmarks except `ocean` less than 1% of the strata were logged due to aliasing. For `ocean` however, 64% of the strata logged are due to false positives. This is because of the large number of dirty block evictions encountered during execution, resulting in heavy use of the bloom filters for the benchmark. Future work should be able to reduce this by improving the bloom filter approach used.

7.3 Bandwidth Overhead

We also collected results for the overheads of our approach in terms of communication bandwidth. For calculating this overhead, we computed how many extra bytes have to be transmitted on the bus to allow us to log the shared memory dependencies. We compute

how many bytes are transmitted on the bus due to read and write misses and the coherence messages associated with it. For the P2P approach, the overhead comes from the extra instruction count piggybacked on the write update reply message and invalidation reply message. The overhead is about 10% extra bandwidth. For our Directory approach, the overhead consists of the memory operation counts piggybacked on the messages sent to the directory as a result of the write misses, read misses and write evicts. Also, the coherence replies from the exclusive owners in response to data fetch request need to be piggybacked with the dependence bit. The overhead is about 12% extra bandwidth, slightly over what P2P requires.

In the snoop-based system, the coherence reply and request messages are piggybacked with one additional bit that instructs the processor nodes whether to log a stratum or not. In addition, before paging, an additional message is broadcast on the bus instructing all the processor nodes to log a stratum. For the programs we examined, we found that these do not incur any appreciable communication overhead - both in terms of number of bytes communicated and in terms of the number of messages communicated.

7.4 Scalability

We finally point out some aspects regarding the scalability of our approach. Note that the number of entries in a strata log is proportional to the number of processor nodes and not threads. As a result, logging overhead will scale linearly with the number of processor nodes in the system. However, for our directory approach, since we clear the dependence bits of only the dependent processor nodes, our transitive optimization to reduce the number of strata logged may not be as efficient when the number of processors increase. In P2P, as the number of processor nodes increase, the number of point-to-point dependencies also potentially increase. In our case, the size of each stratum increases with the number of processors, but so does the benefit of our transitive optimization since it applies across all processor nodes and not just the dependent processors.

8. Conclusion

Providing hardware support for improving software quality and reliability will be increasingly important with future processing trends. Future processors will be multi-core encouraging software developers to use multi-threaded programs to extract parallel performance out of them. Our mechanism for shared memory dependency logging allows deterministic replay debugging for these multi-core systems.

We proposed a new logging mechanism called *Strata*. A stratum is logged across all of the processors every time a shared memory dependency needs to be captured. We log a stratum for every RAW and WAW dependency, and WAR dependencies are determined through offline analysis. A stratum provides a strict time ordering between memory operations that occurred before and after the stratum across the processors. We found that the strata log is 5.8x smaller without compression and 12x smaller with compression than the log used in the previous point-to-point logging approach [13, 6]. Another advantage is that our strata approach requires less hardware than the point-to-point approach. In addition, based on the notion of strata, we were able to design a shared memory dependency logging solution for snoop-based architectures, which the previous proposals did not address.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on this paper. This work was funded in part by grants from Intel and Microsoft.

References

[1] D. F. Bacon and S. C. Goldstein. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206. ACM Press, 1991.

[2] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, Welches, Oregon*, 1998.

[3] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, 36(4):471–482, 1987.

[4] C. K Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[5] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[6] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd Annual International Symposium on Computer Architecture*, June 2005.

[7] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[8] M. Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *International Symposium on High-Performance Computer Architecture*, Feb 2005.

[9] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[10] S. Sethumadhavan, R.Desikan, D.Burger, C.R.Moore, and S.W.Kecler. Scalable hardware memory disambiguation for high ilp processors. In *International Symposium on Microarchitecture*, 2004.

[11] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared-memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002.

[12] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[13] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, 2003.