Computer Sciences Department

Karma: Scalable Deterministic Record-Replay

Arkaprava Basu
Jayaram Bobba
Mark D. Hill

Technical Report #1680

October 2010

Karma: Scalable Deterministic Record-Replay

Arkaprava Basu[$], Jayaram Bobba*[1] and Mark D. Hill[$]
basu@cs.wisc.edu, jayaram.bobba@intel.com, markhill@cs.wisc.edu

[$] Dept. of Computer Sciences
University of Wisconsin-Madison

* Intel Corporation

Abstract

Recent research in *deterministic record-replay* seeks to ease debugging, security, and fault tolerance on otherwise nondeterministic multicore systems. The important challenge of handling shared memory races (that can occur on any memory reference) can be made more efficient with hardware support. Recent proposals record how long threads run in isolation on top of snooping coherence (IMRR), implicit transactions (DeLorean), or directory coherence (Rerun). As core counts scale, Rerun's directory-based parallel record gets more attractive, but its nearly sequential replay becomes unacceptably slow.

This paper proposes *Karma* for both scalable recording and replay. Karma builds episodic memory race recorder using a *conventional* directory protocol and records order of the episodes as a directed acyclic graph. Karma also enables extension of episodes even after some conflicts. During replay, Karma uses *wakeup* messages to trigger a partially ordered parallel episode replay. Results with several commercial workloads on a 16-core system show that Karma can achieve replay speed (a) within 19%-28% of native execution speed without record-replay and (b) four times faster than even an *idealized* Rerun replay. Additional results explore tradeoffs between log size and replay speed.

1 Introduction

Today's shared-memory multiprocessors are not deterministic. The lack of repeatability makes it more difficult to do debugging (because bugs do not faithfully reappear on re-execution) [43], security analysis (attacks cannot be exactly replayed) [10], and fault tolerance (where a secondary set of threads attempts to mimic a primary set to detect faults) [24]. Moreover, dealing with multiprocessor nondeterminism—heretofore limited to a few experts—is now a concern of many programmers, as multicore chips become the norm in systems ranging from servers to clients to phones and the number of cores scales from a few to several to sometimes many.

To this end, researchers have explored software and hardware approaches for a *two-phase deterministic record-replay* system [10, 17, 22, 27, 30, 34, 41, 42]. In the first phase, these systems *record* selective execution events into a *log* to enable the second phase to deterministically *replay* the recorded execution.
A great challenge for record-replay is handling shared *memory races* that can potentially occur on any memory reference, while other events, such as context switches and I/O can easily be han dled by software [10, 22, 28]. Early hardware proposals for handling memory races [41, 42] record when threads *do* interact, but require substantial hardware state to make log sizes smaller.

Three recent hardware race recorders reduce this state by instead recording when threads *don't* interact: *Rerun* [17], *DeLorean* [27] and *Intel Memory Race Recorder (IMRR)* [34]. Let an *episode* (or *chunk*) be a series of dynamic instructions from a single thread that executes without conflicting with any other thread. All three recorders use Bloom filters [5] to track coherence events to determine when to end episodes.

These recorders assume different coherence protocols that affect their scalability to many-core chips and complexity of implementation:

- IMRR assumes broadcast snooping coherence and proposes globally synchronized chunk termination among the cores for better replay speed. IMRR reliance on broadcast and globally synchronized operation limits its scalability.
- DeLorean relies on BulkSC/Bulk's [6, 7] non-traditional broadcast of signatures to commit/abort implicit transactions and a centralized arbiter to record and replay chunk order. Thus DeLorean demands *completely new* coherence protocol and support for implicit transactions to make its scheme for deterministic record-replay feasible.
- Rerun operates with relatively minor changes to more conventional point-to-point directory protocol that allows scalable recording while demands minimal hardware extension.

Thus, going forward, Rerun's approach seems most promising as it is scalable to chips with many cores and to systems with multiple sockets, while requires moderate changes to conventional hardware. During replay, however, Rerun does *not* scale, because its replay is nearly sequential due to its use of Lamport scalar clocks [19]. Fast, parallel replay can expand the applicability of deterministic record/replay systems, which in turn, can further justify deploying them. Fast replay is valuable for scenarios that include:

- In *security analysis*, fast replay can help quick analysis of an attack and allow urgent fix to critical security flaws. A quick replay, even when the attack is underway, can help to trace the attacker [10].
- In *fault tolerance*, where one might wish to maintain availability of a critical primary server in presence of faults, a secondary server following the primary, needs to quickly replay primary's execution to provide hot backup[24].

1. Work performed while at University of Wisconsin-Madison

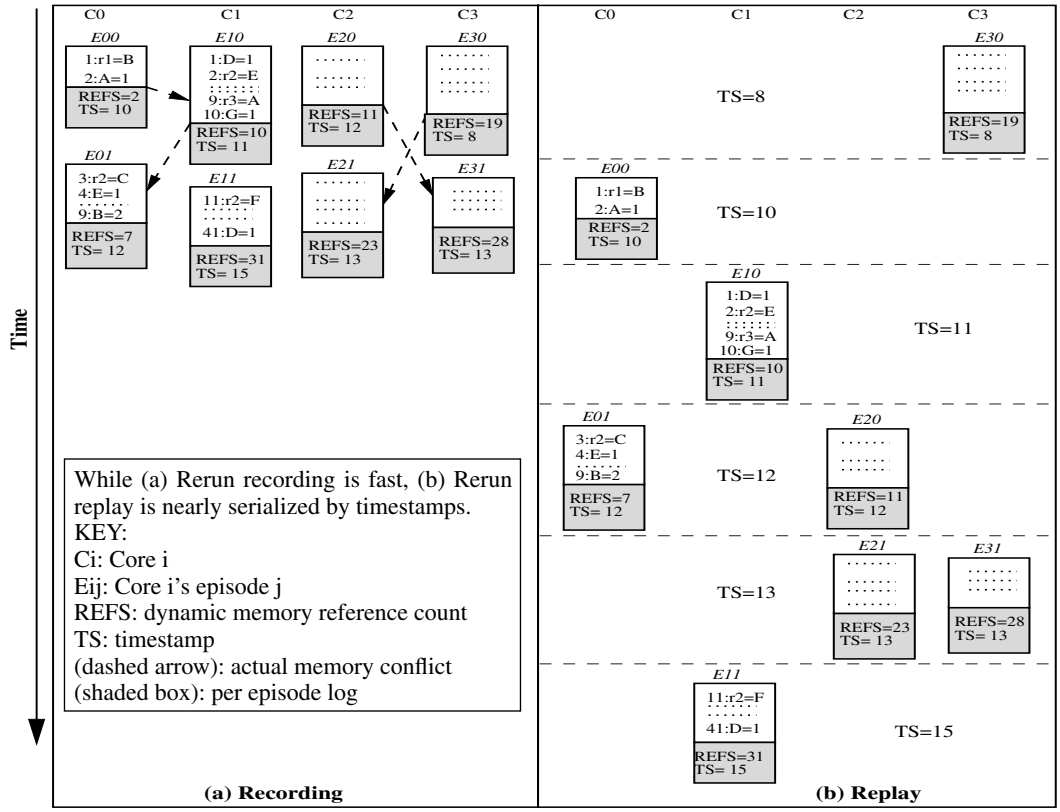

Figure 1. Rerun's Record and *idealized* Replay

- For classic use of *debugging*, deterministic record/replay's utility will decline if scaling to 16, 32 or more cores, requires a sequential replay that is at least 16X, 32X or more slower. Replaying for small intervals of time may be acceptable, but the situation quickly worsens if replay for longer intervals and/or large number of cores are needed.

This paper proposes Karma for both scalable recording and replay, that minimally extends conventional directory coherence protocol. Karma's proposed novel episodic memory race recorder/replayer records the order of episodes as a *directed acyclic graph (DAG)*. Karma also extends lengths of episodes that conflict during recording by ensuring that they do not conflict during replay. During Karma's replay, special *wakeup* messages (like coherence acknowledgment messages) trigger parallel replay of independent episodes.We also show how to extend Karma from sequential consistency to TSO, sufficient to implement the x86 memory model.

We evaluate Karma on a 16-core system and find that: (1) Karma can achieve replay speed within 19-28% of native execution with no-record-replay and about 4 times faster than even *idealized* Rerun's replay. (2) Karma's log size is similar to Rerun's, but (3) can be made smaller for uses that can tolerate slower replay.

The following sections review related work, especially Rerun (Section 2), provide the insights behind Karma (Section 3), describe a Karma hardware implementation (Section 4), review evaluation methods (Section 5), present experimental results (Section 6) and conclude (Section 7).

2 Related Work and Rerun Review

2.1 Related Work

Classic all-software solutions to deterministic multiprocessor replay exist [11, 22], but results show that they do not perform well on workloads that interact frequently. Three recent, promising approaches seek to reduce recording overhead, but consequently make replay more difficult. Park et al. [33] record partial information and retry replay until successful, while Altekar and Stoica [2] seek only to replicate a bug, not an exact replay. Lee et al. [23] seeks to log minimal information but uses online replay on spare cores to validate whether logged information is sufficient to guarantee *output* deterministic replay.

Architecture researchers have focused on solutions that use hardware, at least for memory race detection. Bacon and Goldstein [3] recorded all snooping coherence transactions, which produced a serial and voluminous log. Xu et al.'s Flight Data Recorder (FDR) [41, 42] created a distributed log of a subset of memory races, not implied by other races, but required substantial state with each core. Bugnet [31] shows how to enable record-replay by recording input values rather than memory race order. Strata [30] uses global strata to reduce this state, but does not scale well to many cores [17]. ReEnact [35] allowed deterministic reproduction of a recent buggy execution with Thread Level Speculation (TLS) support. As previously discussed, DeLorean, Rerun, and IMMR largely eliminate FDR's filtering state by focusing on when cores operate independently. More recently, Timetraveller [39] improved upon Rerun to reduce its log size further by delaying ending of episodes in Rerun. Herein we propose Karma to improve Rerun's replay speed, and we expect that Karma's improvements will apply to Timetraveller as well.

Importantly, Capo [28] discusses how to virtualize hardware deterministic replayers—including FDR, Rerun, and DeLorean—so that different parts of a machine can be in different modes: recorder, replay, or none. Fortunately, Karma, can also be virtualized with Capo.

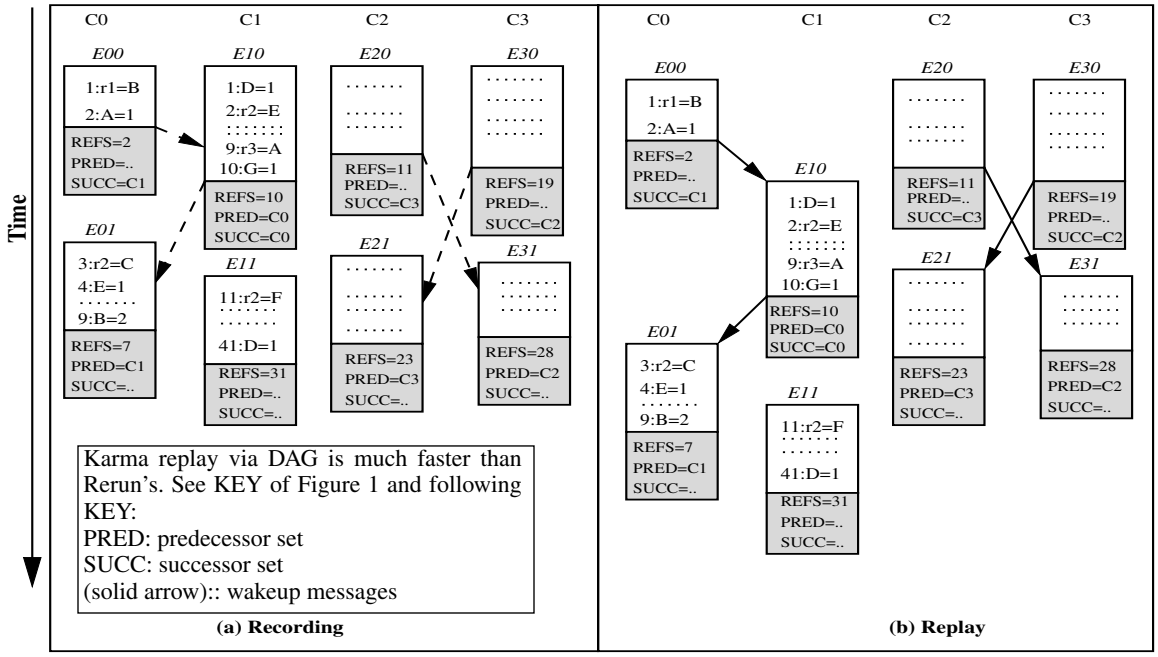

Figure 2. Karma's DAG-based Record and Replay with Rerun's Episodes

Finally, there have been several recent efforts on obtaining deterministic execution, wherein a multithreaded program with a fixed input always executes the same way [4, 9, 32]. Somewhat related is Yu et al.'s work [44] to constrain production software runs to the set of interleaving observed during testing. While promising, these approaches are not (yet) generally adopted.

2.2 Rerun Review

We review Rerun here to better enable Section 3 to show how Karma supersedes it, even as both modestly extend conventional directory cache coherence protocols.

Record. Rerun dynamically breaks each core's execution into *episodes* during which a core does *not* interact with other cores. Rerun ends an episode when memory references of an episode conflicts with a concurrent episode on another core. It can ends episodes early, e.g., due to false conflicts, L1 cache evictions, or context switches. Rerun orders episodes with the timestamps based on a Lamport scalar clock [19]. Rerun's global log is a distributed collection of per-core logs. Each per-core log captures a core's sequence of episodes with each episode's size in dynamic memory references (REFS) and Lamport scalar clock timestamp (TS). Figure 1(a) illustrates a Rerun recording, after threads at each core executed for some time initially. In Figure 1(a), when during episode E10, core C1 tries to read memory block *A,* a coherence intervention message is sent to core C0, which had written the same address as part of episode E00. This prompts C0 to end episode E00, as it detects a conflict and attaches its own timestamp in the coherence reply (dotted directed edge in Figure 1(a)). After receiving the coherence reply, core C1 adjusts the timestamp of episode E10 accordingly to capture the fact that E10 must be ordered after E00 during replay.The proposed Rerun implementation uses per-core read and write Bloom filters to detect when to end episodes and piggybacks timestamps on coherence response messages to capture the causal ordering among the episodes.

Replay. Rerun advocates software-based fully sequential replay of episodes in increasing order of their timestamps. In theory, however, scalar timestamps allow some parallelism, where episodes with the same timestamp can be replayed concurrently. We illustrate this *idealized* Rerun replay (non-sequential) in Figure 1(b). On one hand, it allows episodes E21 and E31 to be replayed concurrently. One the other hand, Lamport scalar clocks unnecessarily orders many *independent* episodes (e.g., E20 with episodes from cores C0 and C1).

3 Karma Insights: Replaying Episodes in Parallel

As multi-threaded programs scale to more cores, replay must be parallelized otherwise it can become arbitrarily slow, limiting the utility of record-replay for online uses (e.g., fault tolerance, security analysis) and eventually debugging. To this end, this section introduces insights into Karma's parallel replay with both (a) ordering episodes with DAG and (b) extending episodes. While we present how Karma orders the execution in the cores, Karma—like FDR, Rerun, and DeLorean—can be virtualized by Capo [28].

3.1 Key Idea 1: Using a Directed Acyclic Graph to Order Episodes During Replay

The first key idea behind Karma is simple: *Use a directed acyclic graph (DAG) rather than scalar timestamps to partially order episodes during replay.* DAGs are well known to allow much greater parallelism than scalar timestamps and have been used in an offline analysis of replay speed potentials of deterministic recording schemes [34]. For ease of exposition, we first show the value of using a DAG by pretending that Karma's recording breaks the execution into exact same episodes as Rerun did in Figure 1, and then, in Section 3.2, present a second innovation that allows Karma to have longer episodes than Rerun permits.

To this end, Figure 2(a) illustrates how Karma can record memory dependencies among cores by triggering episode formation with DAG edges to successor episode(s). Karma's distributed log resembles Rerun's log with timestamps replaced by DAG edges (represented as PRED/SUCC sets explained below).

Figure 2(b) illustrates the parallelism of Karma's replay wherein successor episodes execute after their predecessors without other *artificial* ordering constraints. Importantly, this enables a parallel replay that is much faster than even Rerun's idealized replay. For example, while Rerun ordered episode E20 with *independent* episodes of cores C0 and C1 (Figure 1(b)), Karma's replay leaves episode E20 unordered with respect to the episodes of cores C0 and C1 (Figure 2(b)), facilitating more replay parallelism.

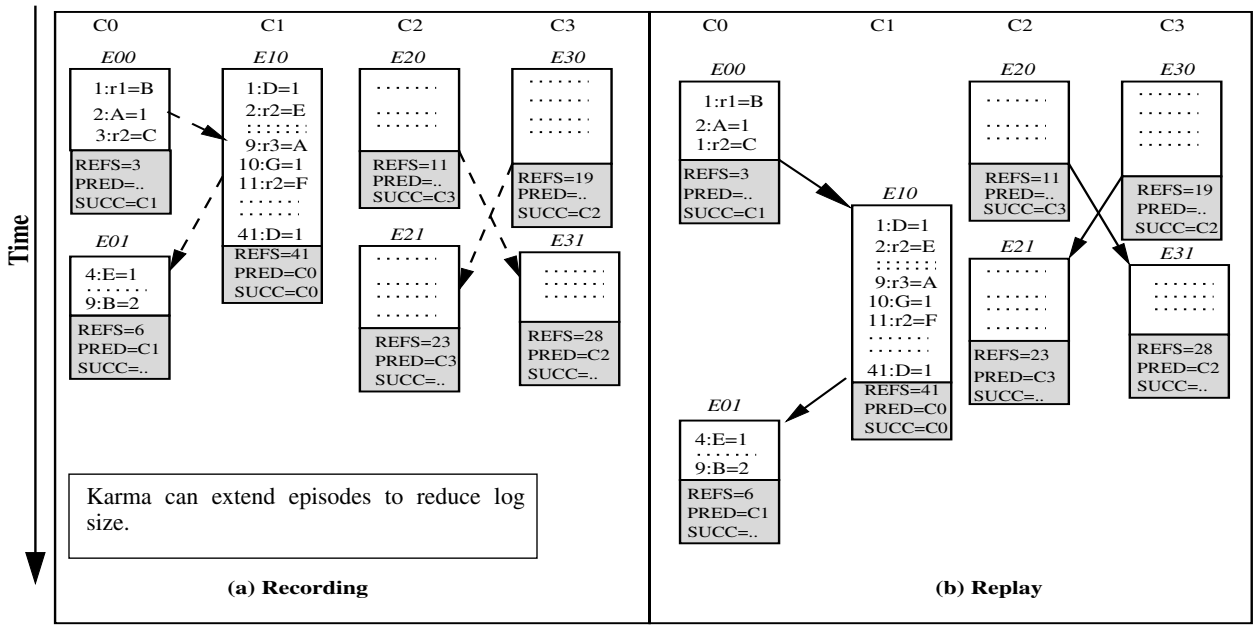

Figure 3. Karma's Record and Replay with Extended Episodes

While the idea of using a DAG is simple, it is less simple to determine how to represent DAG edges to successor episode(s). For fastest replay, the DAG edge representation should facilitate an episode waking up the successor episode(s) quickly. Moreover, for low recording overhead, it should be fast to create during recording and compact to log. Using integer episode identifiers, as in a software representation of DAG edges, is a poor representation, as we see no way for replay to avoid indirecting through memory to determine the successor(s). Using these episode identifiers would also have severe negative impact on log size.

As discussed more fully in Section 4.3, to efficiently record the DAG edges, Karma actually represents DAG edges with *predecessor* (PRED) and *successor* (SUCC) *sets* that name the cores of the predecessor and successor episodes respectively. During recording, these sets are populated from coherence traffic and then logged. During replay, a core awaits a wakeup message from each predecessor before beginning an episode and sends a wakeup message to each successor after completing an episode.

3.2 Key Idea 2: Extending Rerun's Episode

The second key idea behind Karma is subtle: *Concurrent episodes must not conflict during replay, but may conflict during recording.* In contrast, Rerun, DeLorean and IMRR always ends episodes when they conflict during *recording*. For example in Figure 1(a) for Rerun, core C0 ends episode E00 when it gives block A to core C1 for episode E10. In Figure 2(a), we show Karma behaving similarly, but this is not necessary. More recently, Timetraveller [ref] which improves upon Rerun's log size uses *post-dating* of scalar timestamps to also allow growing episodes even after some conflicts.

In contrast, as shown in Figure 3(a), Karma continue recording in episode E00 even as it conflicts with episode E10, as long as it orders E00 before E10 in the log. During replay, conflicting episodes E00 and E10 will not be concurrent, because the log entries will ensure that the end of E00 precedes the beginning of E10. In similar fashion, core C1 can cover its execution of 41 references with one episode E10 (Figure 3(a)), rather than two episodes E10 and E11 (Figure 2(a)). Beside the restriction discussed below, a core is not required to end a episode when either it (a) provides a block to another core or (b) obtains a block from another core.

On one hand, this optimization *seems* too good to be true. Perhaps the authors of Rerun and DeLorean missed it, because they appear to be inspired by transactional memory systems [15, 21] that usually abort when concurrent transactions conflict in an execution (as there is no distinction between recording and replay). Fortunately in *Dependence Aware TM*, Ramadan et al. [36] showed that conflicting concurrent transactions can all commit, provided that they are properly ordered. For example, they allow core C0's transaction T to pass a value to core C1's concurrent transaction U (and both commit) as long as T is ordered before U. Karma exploits a similar idea for episodes. Both are inspired by the greater freedom of conflict serializability over two-phase locking [12] and value forwarding among "episodes" in some thread-level-speculation systems (e.g., [13, 37]).

On the other hand, full exploitation of the optimization *is* too good to be true. As depicted in Figure 3(a), a problem occurs when the core C0 later *attempts* to order E00 *after* core C1's episode E10 because of conflict in block E (memory reference 4 of core C0), but E00 was previously ordered *before* E10 due to block A (or conversely a core seeks to order an episode *before* another episode previously ordered *after*). Karma cannot do this without adding a cycle to the DAG, which is not allowed, as it would make ordering replay impossible. Instead, Karma always ends episode E00, begins episode E01 (with memory reference 4 as its first reference), and orders E01 after E10 of core C1.

Karma detects the possibility of cycle formation in the recorded DAG using Lamport scalar clock based timestamps [19] (but never logs them). Karma ends an episode when it receives a timestamp greater than the timestamp of the current episode. This ensures that the order of episodes is acyclic and can be replayed properly. Since Karma does *not* log timestamps, they can not serialize replay and the sole purpose of this timestamp is to dynamically detect possibility of cycles while recording.

Finally, Karma enables a *tradeoff between log size and replay parallelism*, similar to one found in other record-replay systems [27,42]. Growing longer episodes has two effects. First, larger episodes mean fewer episodes to cover an execution. This makes log size smaller. Second, longer episodes make replay less parallel and slower. This is because during replay the *end* of a predecessor epi-

sode happens before the *beginning* of a successor episode. For example, earlier we saw that Karma could cover core C1's execution of 41 memory references with one episode (Figure 3(a)) rather than two (E10 and E11 in Figure 2(a)). In Figure 3(b), we however observe that during replay, this means that episode E01 can only start execution after the merged bigger episode E10 completes its execution. For this reason, as we will find in Section 6, there is value in bounding the maximum episode size to balance log size and replay parallelism.

3.3 A Sketch of Karma Operation

This section sketches Karma's basic operation for recording and replay, but leaves details for Section 4.

Record Sketch. During recording, Karma, grows episodes and passes timestamps on coherence response messages. Each core grows its episode until it receives a timestamp greater than its current timestamp (or a maximum size is reached, etc.). This indicates possibility of cycle in the DAG. At this point, it ends its episode, saves the corresponding predecessor/successor set for logging, and begins a new episode. When responding with a timestamp, a core sends its current timestamp for a block that matches in its read/write filter or its previous timestamp otherwise. For implementation reasons discussed later, a Karma core keeps the timestamp and predecessor/successor sets for both its immediately previous and current episodes. When an episode ends at a core, it logs the memory reference count, predecessor and successor set of the immediately previous episode, but *never* logs the timestamp.

Replay Sketch. During replay, a Karma core repeats four steps. (1) Read the predecessor/successor (PRED/SUCC) sets and reference count REFS for its next episode. (2) Wait for wake-up messages from each core in the episode's predecessor set. (3) Execute instructions for REFS memory references. (4) Send a wakeup message to each core in the successor set.

Online Replay? While we present the record and replay phases as separate, applications like fault tolerance may wish to "pipe" the log from recording to a concurrent replay. Karma's faster parallel replay makes this online replay more promising, but we leave detailed design issues to future work.

4 Implementing Karma

While the previous section presented the ideas behind Karma, this section presents a concrete hardware implementation and addresses additional issues.

4.1 Example Base System

We assume a base system as illustrated in Figure 4 with parameter values from Table 1. It is a multicore chip with private writeback L1 caches, shared multibanked L2 and a MESI directory protocol.

4.2 Karma Hardware

As Figure 4 depicts, Karma adds eight registers (148 bytes) to each core: 128-byte address filter (FLT) (combining Rerun's read/write filters), 4-byte reference count (REFS), and for both the previous and current episodes, there are predecessor sets (PRED0 and PRED1), successor sets (SUCC0 and SUCC1) and 4-byte timestamps (TS0 and TS1). For 16 cores, all sets can be represented with 2-byte bit vectors, while more scalable representations are possible as many episodes have one or two predecessor or successor.

Karma assumes L2 cache blocks include a directory that tracks where a block is cached in the L1s, L1 cache shared replacements are silent, and L1 writebacks continue to remember the previous owner. Section 4.6 will discuss additional issues due to L1 and L2 caches being finite. Karma passes timestamps on coherence response messages. Karma adds a single bit called *previouslyOrdered* in coherence response message, to be explained in Section 4.3. For supporting replay, Karma adds wakeup messages whose only payload is a source core identifier.

4.3 Predecessor and Successor Sets

Table 1: Base System Configuration

Cores	16, in-order, 3GHz
L1 Caches	Split I&D, Private, 32K 4-way set associative, write-back, 64B lines, LRU replacement, 3 cycle hit
L2 Cache	Unified, Shared, Inclusive, 16M 8-way set associative, write-back, 16 banks, LRU replacement, 21 cycle hit
Directory	Full bit vector in the L2
Memory	4G DRAM, 300 cycle access
Coherence	MESI Directory, silent replacements
Consistency Model	Sequential Consistency (SC) (with extension to TSO in Section 4.7)

This subsection discusses some subtle issues for how and why Karma represents DAG edges between episodes as predecessor and successor sets between cores. We implement each set with a 2-byte bit vector, but larger systems can use other encodings since most of these sets have just one or a few elements.

Since predecessor and successor sets can only record a single edge from/to each other core, we take special care to avoid recording a second edge between the same two cores (Figure 5). When sending a message that would constitute a second outgoing edge from an episode to the same other core, we set the *previouslyOrdered* bit in the coherence reply message to indicate that this message does not represent an edge, as depicted in Figure 5(a) and (b). It is correct to elide this edge, as it is redundant because of the previous edge to this core. On receiving a message that would be the second incoming edge from another core, we end the receiving core's episode, start a new episode, and add the edge to the otherwise empty new predecessor set (Figure 5(c)). This is correct, since cores can always end episodes early. On receiving a request message from a core

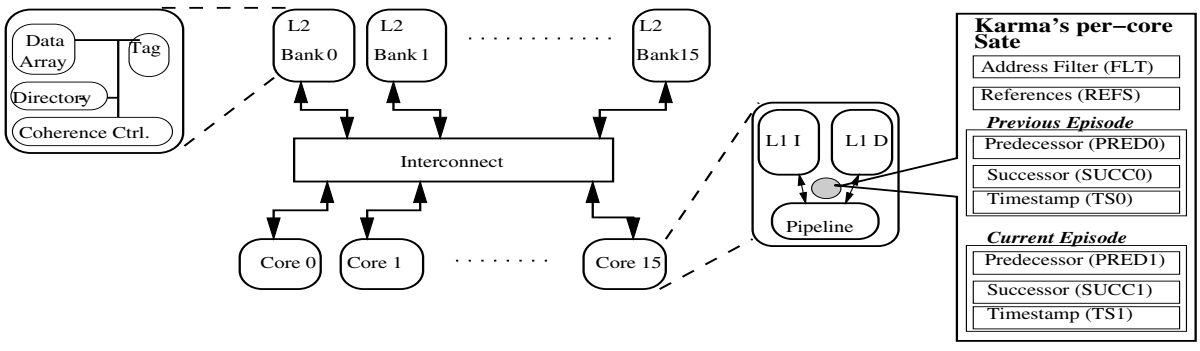

Figure 4. Base System Configuration with Karma's State per Core

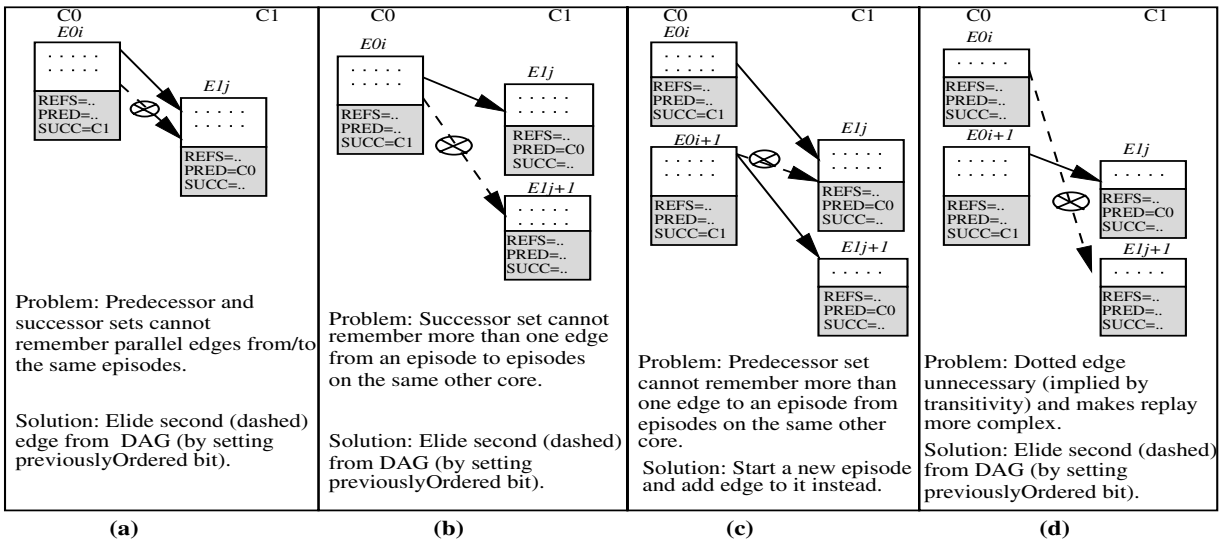

Figure 5. Subtle Implementation Issues Regarding Predecessor and Successor Sets

already ordered after this core's current episode, this core responds with the *previouslyOrdered* bit set so that this message is also not a DAG edge, as depicted in Figure 5(d). This action is correct because the missing edge is implied by transitivity [41].

Karma's approach for representing DAG edges leads to a convenient invariant during replay (Section 4.5): when a core receives a wakeup message from core *req,* the message pertains to the receiving core's next episode whose predecessor set includes core *req.* This allows the wakeup message to physically name a core and yet have the edge be applied to a specific episode as in Figure 3(b).

4.4 Karma Recording

As depicted in Figure 6, the key to Karma recording is what actions Karma takes when a core/L1 *sends* a data response or acknowledgement (left side) and *receives* data or an acknowledgement in response to a coherence request it has made (right side). The top of Figure 6 repeats the Karma state from Figure 4.

During recording, each core sometimes sends a coherence reply (data and acknowledgement) in response to coherence request from another core *req*. The core first tests whether core *req* is already an element of SUCC1. If it is, the outgoing message's *previouslyOrdered* bit is set, so that the message does not create an edge in the DAG (Section 4.3) and no other actions are needed.

Otherwise, the core examines whether its address filter contains the message address (or a false positive). If so, then the core associates the outgoing edge with its current episode. It sets the message's timestamp to TS1 and *previouslyOrdered* bit to false. It then adds core *req* to SUCC1. If the filter does not match, the core associates the message with its previous episode and takes corresponding actions using TS0 and SUCC0. This is correct, because if a block is not touched by the current episode it was touched no later than the previous episode at that core.

During recording, a core executes instructions, which sometimes generate cache misses and coherence requests. Upon receiving a coherence response message (data or acknowledgement) from core *src*, a core may or may not take any actions for recording (Figure 6(b)). In particular, if the incoming message's *previouslyOrdered* bit is set, no action is needed, because the message comes

State per core

```
REFS0: Memory Reference count
       for previous  episode
REFS1: Memory Reference count
       for current episode
FILTER: Address filter
```

```
TS0: Timestamp of previous Episode
TS1: Timestamp of current Episode
PRED0: Predecessor set of previous Episode
SUCC0: Successor set of previous Episode
PRED1: Predecessor set of current Episode
SUCC1: Successor set of current Episode
```

Coherence Message Structure

```
src/req: Source or Requestor core id
previouslyOrdered: need an Edge in DAG?
```

```
dst: Destination
TS: Timestamp
```

```
Addr: Address
Payload: Data etc.
```

```
Action on sending data/Ack reply:
 receive_request_message(i_msg)
 if(SUCC1 contains i_msg.req) {
       /* No new edge in DAG neded */
      o_msg.previouslyOrdered = true
  } else {
       if(FLT contains i_msg.Addr){
           /* Current episode*/
            o_msg.TS = TS1
            o_msg.previouslyOrdered = false
            SUCC1.set(i_msg.req)
        } else {  /* Previous episode */
            if(SUCC0 contains i_msg.req) {
               /* No new edge required */
              o_msg.previouslyOrdered = true
            } else {
               o_msg.TS = TS0
               o_msg.previouslyOrdered = false
               SUCC0.set(i_msg.req)
            }
        }
  }
 /* Fill in other fields in o_msg according
   to coherence protocol */
 send_response_message(o_msg)
```

(a)

```
 Action on receiving data/Ack i_msg:
if(i_msg.previouslyOrdered == true) {
     /* Do nothing */
 }else{
     if((SUCC1 not empty && i_msg.TS>=TS1)
          || PRED1 contains i_msg.src){
          /* End episode */
         /* Log previous episode */
          write_to_log(REFS0,PRED0,SUCC0)
       /* Move current episode to previous*/
          TS0=TS1
          REFS0=REFS1
          PRED0=PRED1
          SUCC0=SUCC1
          /* Set up new episode */
          Clear PRED1,SUCC1 and FILTER
          PRED1.set(i_msg.src)
          REFS1=0
          TS1 = max(i_msg.TS+1,TS0+1)
       } else {
        /* Update the current episode */
          TS1 = max(i_msg.TS+1, TS1)
          PRED1.set(i_msg.src)
       }
     }
```

(b)

Figure 6. Karma's Recording Algorithm (at each core)

```
State per core:
PRED1: Predecessor set of current episode (executing/waiting)
SUCC1: Successor set of current episode (executing/waiting)
REFS1: Count of residual memory references of current episode

Wakeup message structure:
src: Source                          dest: Destination

Before the start of a new episode:
   /* (1) Get next log entry */
   read_from_log(REFS1,PRED1,SUCC1)
   while(PRED1 not empty) {
     /* (2) Wait until all required wakeup messages arrive */
     when receive_wakeup_message(i_msg) {
        if(PRED1 contains i_msg.src) {
             PRED1.unSet(i_msg.src)
        }
     }
   }
   /* (3) Execute  the new episode */
   while(REFS1 != 0) {
      Execute instructions
      if(memory reference) {
          REFS1 = REFS1 - 1
      }
   } /* Finished episode execution*/

   /* (4) Send out wakeup messages */
   For each element P in SUCC1 {
        o_msg.src = own_core_id
        o_msg.dest = P
        send_wakeup_message(o_msg)
   }
```

Figure 7. Karma's Replay algorithm (at each core)

from a core whose current or previous episode was already ordered with respect to this core's earlier or current episode.

If episode ordering is required, the incoming message may cause the current episode to end for two reasons. First, the episode ends if SUCC1 is not empty and the message's timestamp is greater than the current episode's timestamp. This is done to prevent cycles in the DAG. Second, the episode ends on incoming message from core *src* that is already in the current episode's PRED1.

To end an episode, a core logs the previous episode's memory reference count and the predecessor/successor sets, copies the current episode's information to the previous one's, and then initializes the new current episode's values. In particular, the timestamp update follows Lamport scalar clock rules, the filter is cleared, the successor set made empty, and predecessor set made to contain only the message source (core *src*). The timestamp is *not* logged and thus has no role in replay.

4.5 Karma Replay

During replay, a Karma core repeat four steps, depicted in Figure 7.

(1) When a core is ready to start a new episode, it reads the predecessor/successor (PRED1/SUCC1) sets and reference count REFS1 for the next episode from its per-core log. These values are stored in the same special registers as used in recording. Replay on this core is complete when its log is empty.

(2) The core waits for wakeup messages from each core in the episode's predecessor set PRED1. When the core has received a message for all cores originally in PRED1, it moves to the next step.

(3) The core executes instructions of the episode, decrementing REFS1 on each dynamic memory references, and stops execution when the episode REFS1 is zero and the episode is complete.

(4) The core sends a wakeup message to each core in its successor set SUCC1. When complete, the core goes back to step (1).

Karma's replay algorithm counts architectural memory references, but never micro-architectural events, such as cache misses. Thus, Karma replay does not require the same caches or cache state as was present during Karma recording.

The description above acts as if the wakeup messages arrive only during step (2), whereas they can actually arrive at any time. We implement a simple replayer that just buffers early messages. A more complex replayer could "pipeline" episodes by reading the next log entry early and gathering wakeup messages for the next episode while the current episode is still executing.

More subtly, wakeup messages for future episodes can arrive earlier than ones needed for the next episode(s), theoretically filling up any fixed sized message buffer. Fortunately, since the only information that must be remembered about a wakeup message is its source core identifier, a core can remember up to 8 wakeup messages per core (128 total) using a three-bit counter for each of 16 cores (6 bytes total). Moreover, these buffer counts can be made unbounded using known "limitless" techniques [8] that maintain rare overflow counts in software.

4.6 Effect of Finite Caches

Heretofore we assumed infinite L1 and L2 caches, but real systems have finite caches. Here we extend Karma to handle L1 and L2 cache replacements (from 'Shared' and 'Exclusive') and writebacks (from 'Modified') [38]. Many solutions are possible (and could be an entire paper). Assume that a block is evicted by core C0 at episode E00 and next used by core C1 in episode E19. In all cases, episode E00 must be ordered before episode E19.

L1 Evictions. Karma handles L1 replacements and writebacks mostly like FDR [41]. There are three possible cases during L1 eviction that require attention. First, a shared replacement by C0 is silent. A subsequent miss by C1 will send an invalidation to C0 whose acknowledgement message will order episode E19 after C0's current episode which is (long) after episode E00. Second, a writeback by C0 does not reset the block owner field at the L2, much like LogTM's sticky states [29] and FDR [41]. As in the first case, a subsequent miss by C1 will send an message to C0 whose acknowledgement message will order episode E19 after C0's current episode which is after episode E00. Third, C0 could have written back the block and one or more other cores read it. Here Karma, extends the L2 directory by 4 bits (< 1% of a 64-byte cache block) to keep core identifier of the last writer to a block, so that reads can continue to get ordered after C0's current episode that is after E00. This is the same state that a MOESI coherence protocol needs to remember for an owner among sharers.

L2 Evictions. Karma seeks a different solution for L2 evictions, because (a) they are much less common and (b) we wish to add little or no state to main memory. The key idea is to compute a proxy core to order the eviction before any subsequent use. For example, the proxy core for victim block 100 with 16 cores might be 100 modulo 16 = C4. When the L2 seeks to evict block 100 last written by core C0, it will first order the current episode of C0 before the current episode of C4. (Much) later when C1 misses to memory for block 100, the L2 can recompute the proxy C4 and order the current episode of core C4 before the current episode of core C1 which is E19. By transitivity, episode E00 is ordered before E19. Optionally, memory can use a single bit to remember whether a block was ever cached, as we assume in our simulations. Many other solutions are possible, including broadcasting on L2 misses in small systems or augmenting main memory to remember the previous writer if metabits are available.

4.7 Extending Karma to Support TSO (and x86)

Hitherto, Karma implicitly assumed the sequential consistency (SC) memory consistency model [20], but now we show how to extend Karma to total store ordering (TSO) [14, 40]. Unlike SC, TSO exposes (an abstraction of) write buffer for committed writes. Moreover, TSO provides a correct implementation of the x86 memory model [18] that exploits most of the flexibility that x86 allows. We extend Karma by adapting Xu et al.'s TSO solution from the dependence-based RTR [32].

TSO presents challenges as it allows a processor to *commit* a write (store) before a subsequent read (load) (in program order) and yet *order* the write (at logical shared memory) after the read. In practice, this relaxation of write-read ordering is leveraged using a first-in-first-out write buffer to hold writes that are committed but yet not *ordered*. Xu et al. [32] showed that such write buffers can cause their RTR system to record a cycle of dependences and deadlock the replay. To break these cycles, they propose a *order-value hybrid* recorder that detects a *problematic read* (or load) and reacts by recording the value read and not recording the write-after-read dependency that made the read problematic. Specifically, a problematic read is a read that gets its value V from the cache, while one or more earlier committed writes (in program order) are in the write buffer and cache block containing V is invalidated before all earlier writes are ordered. The execution is replayed following the now-acyclic dependencies and "bypassing" values to reads from the log whenever present.

We found that Karma's replayer can also run into a similar situation for the same reasons, but fortunately, Xu et al.'s solution can be extended to episodic record/replay of Karma. Figure 8 illustrates how we can change Karma to allow order-value hybrid recording in the presence of problematic reads in a TSO execution. Assume that the writes from both the cores are ordered after the subsequent reads, so that both reads return the value 0. Both reads will be detected as problematic (with Xu et al.'s detection mechanism), cause the new recorder to simply log the value of each read (as shown in the Figure 8) and to *not* modify the predecessor or successor set to effectively omit logging the dependency (following Xu et al.). Figure 8 shows how logs for both episodes E00 and E11 could include entries that each episode's second instruction (offset 1) obtained value 0. Thus, Karma records a TSO execution that is not an SC execution.

5 Evaluation Methods

We evaluate Karma using the multicore hardware presented in Section 4, except that we study scaling by varying core count: 4, 8, and 16 cores. When doing this, we keep the shared L2 cache size per core constant at 1MB, so the total L2 cache size is 4MB for 4 cores, 8MB for 8 cores, and 16MB for 16 cores.

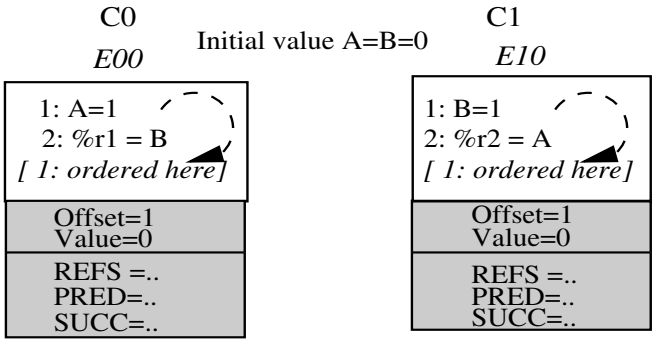

Figure 8. Karma's handling of TSO execution through hybrid order-valu recording.

For comparison purposes, we also evaluated Rerun [17] in the same setup, as it is the closest cousin to Karma, using the code from Rerun's *recorder* provided to us by Wisconsin. More specifically, we compare against an *idealized* Rerun *replayer* (non-sequential) that (a) replays episodes with the same timestamp in parallel (as in Figure 1(b)) and (b) appears to wakeup episode(s) with the next timestamp after the last episode with the current timestamp completes. A practical implementation of Rerun replay would, of course, be slower than this idealized one.

We use the Wisconsin GEMS [26] full system simulation infrastructure, which models an enterprise-level SPARC server running on unmodified Solaris 9 operating system. This simulator uses the Simics [25] full system simulator as front end for the functional part of the simulation and uses the Ruby memory timing model to simulate different hardware platforms. In this work we concentrate on memory race recording and replaying and assumes support for handling DMA, I/O, and external interrupts much like FDR [41] and software layer support much like Capo [28]. To approximate this, we dilate Simics's time to make sure interrupts arrive between the same dynamic instructions during both recording and replay.

We use the Wisconsin Commercial Workload suite [1] to drive evaluation. This workload suite consists of a task-parallel web server (Apache), a java middleware application (Jbb), a TPC-C like online transaction processing (Oltp) workload on DB2, and a pipelined web server (Zeus). We stress-tested Karma implementation with memory race ordering sensitive microbenchmark racey[16].

6 Experimental Results

This section will ask three basic questions and provide the answers summarized here:

Question #1: Does Karma speedup replay? Yes, Karma replay can be 1.4X-7.1X faster than idealized Rerun replay. This translates to a modest 19%-28% slowdown for the replay over the base system without any record/replay in a 16 core system. With fewer cores the slowdown is even less.

Question #2: How can Karma trade off log size and replay speed? By loosening the bound on maximum episode size, Karma can achieve smaller log sizes (e.g., 47%) for situations when slower replay is tolerable (e.g., 21% slower). This presents an effective control knob to trade off the log size versus replay speed, depending upon the requirement of a particular use of deterministic replay.

Question #3: How does Karma's log size compare to Rerun's? Karma and Rerun log sizes are comparable with a 256-memory-reference maximum episode size. As we increase episode size, Karma achieves substantial log size reduction (up to 43%) over Rerun and still replays much faster than Rerun.

6.1 Big Picture: Much Faster Replay While Retaining Fast Recording

Figure 9 displays the most important histograms in this paper. Each histogram provides results for a different benchmark. Each cluster of bars within a histogram represents a different configuration of the CMP. The configuration here is characterized by the number of cores (4, 8, and 16) and total L2 cache size (4MB, 8MB, and 16MB). All histogram heights depict the speedup normalized to the native execution at corresponding CMP configuration with no recording or replay (Base). For example, bars clustered around x-axis point for 16core-16MB configuration are speedup normalized

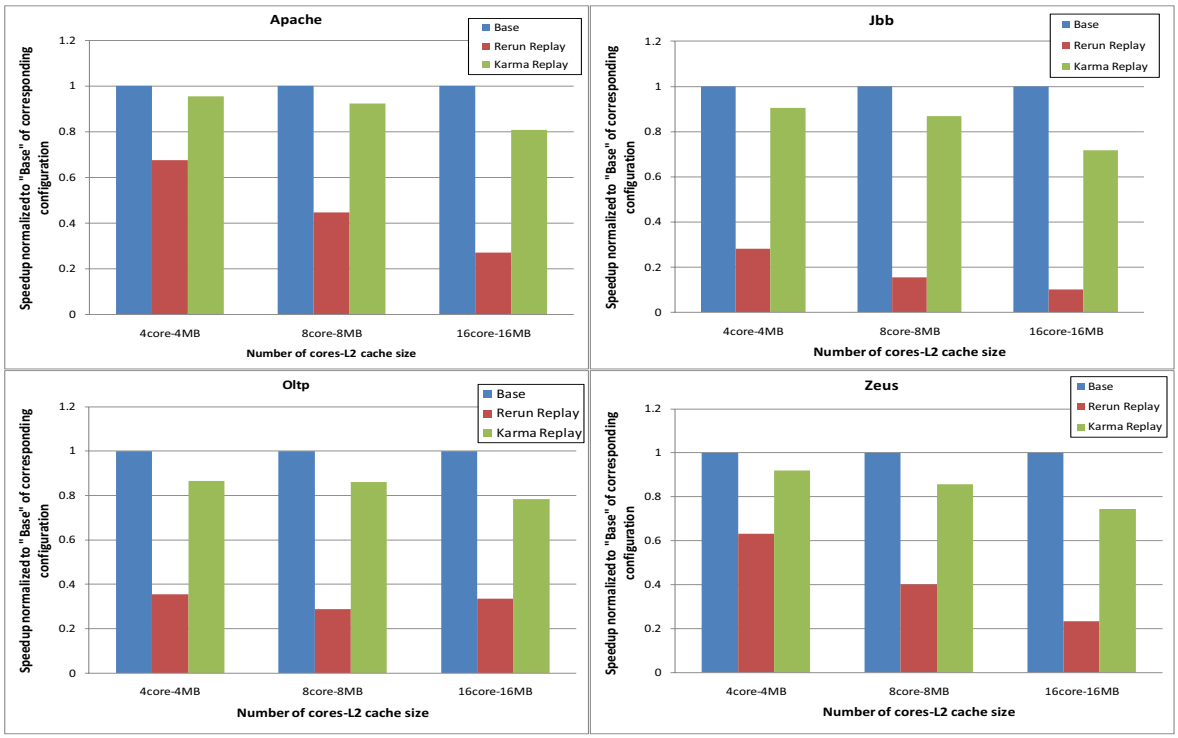

Figure 9. Comparison of Rerun's and Karma's Replay Speed (normalized to Base, 256-reference max. episodes)

to execution for 16core-16MB L2 cache CMP configuration without any record/replay (Base).

Each cluster of bars in a histogram has three bars representing execution with ***no record/replay (Base)***, idealized ***Rerun replay*** and ***Karma replay***. Not shown are Rerun and Karma recording, as they are mostly very close to Base. All results in this section assume a maximum episode size of 256 memory references, which, as we will see later, produces a log size similar to Rerun's. By showing performance comparison of our system for different configuration points for CMP, we tried to demonstrate scalability and applicability of our proposed system under varying CMP configurations.

Question #1: Does Karma speedup replay? Yes, Karma replay can be 1.4-7.1X faster than idealized Rerun replay depending upon the number of cores and the application. For Apache with 4, 8, and 16 cores, for example, Karma replay is 1.4X, 2X, and 3X faster than Rerun replay. This is not surprising, since even our *idealized* Rerun replay still does not expose enough parallelism due to scalar clock based ordering of episodes. Results for Oltp and Zeus are similar, while for Jbb, Karma replay speed is 7.1X of Rerun's. For the 16-core system, Karma replay is only 19%, 28%, 22% and 25% slower than the Base execute (without record/replay) for Apache, Jbb, Oltp and Zeus, respectively. For fewer cores, this slowdown is much less.

Recording. Karma's recording (not shown in the figure) is usually negligibly different from doing no recording/replay (i.e., Base), just like Rerun's recording. We observed for Apache, Jbb and Zeus, Karma recording runs nearly identically (<1% slowdown) with doing no recording (Base). Karma recording also works similarly well for OLTP with 4 and 8 cores, but adds a bit more overhead (~10%) to 16-core OLTP. More importantly, Karma recording scales well when applications scale well.

Bandwidth. Karma sends around 2% more traffic over the interconnect while recording than compared to Rerun. This is because of extending few coherence messages (*previouslyOrdered* bit, etc.) and sometimes because of extra messages. This modest extra bandwidth could affect execution time substantially if the interconnect was near saturation (which it should not be). One would note that Rerun adds around 10% bandwidth overhead on the interconnect over a system with no record/replay.

Sensitivity Analysis. The results above showed that Karma gains are robust with varying core count (4, 8 and 16) and shared L2 cache sizes (4, 8, 16 MB). Additional sensitivity analysis (not shown), confirms that results qualitatively hold during variations, e.g., doubling L1 size, doubling L2 size, and halving memory latency.

6.2 Obtaining Smaller Logs but Slower Replay

Above we showed that Karma replay performance is much better than Rerun's (1.4X-7.1X) for what will turn out to be a comparable log size. Some uses of record/replay, e.g., debugging, may wish to reduce rate of log size growth further, so that, for fixed log size, one can record a longer execution. Karma facilitates this for deterministic record/replay uses that can tolerate somewhat slower replay. This

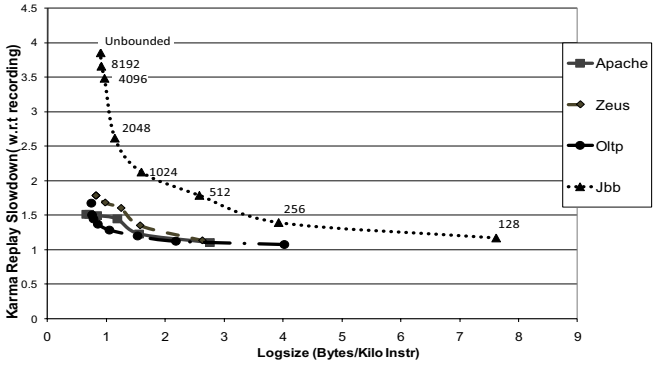

Figure 10. Tradeoff between Log size and Replay Slowdown.

By varying maximum episode size across 128 (right end of each application's lines), 256, 512, ... 8K, and unbounded number of memory references, allow one tradeoff smaller log with some replay slowdown. Displayed results use 16 cores.

might be a good tradeoff when recording is much more common (e.g., always on) than replay (e.g., to investigate a crash).

Figure 9 illustrates Karma log size and replay speed for each of our four applications. The x-axis gives uncompressed log size growth in bytes per thousand instructions. The y-axis gives replay speed normalized to recording speed. For each application, each point on its lines, beginning from the *right*, provides the tradeoff with maximum episode sizes with memory reference counts: 128, 256 (the value used in Section 6.1), 512, ..., 8K, and unbounded.

Question #2: How can Karma trade off log size and replay speed? By loosening the bound on maximum episode size, Karma achieves smaller log sizes but slower replay. For example, increasing the maximum episode size from 256 to 2K references has the following effects: Apache generates 54% smaller log for 23% slower replay, Jbb generates 71% smaller log for 87% slower replay, OLTP generates 60% smaller log for 21% slower replay, and Zeus generates 47% smaller log for 33% slower replay. Three of four benchmarks pay a modest replay slowdown, while Jbb is more sensitive.

Question #3: How does Karma log size compare to Rerun's? Karma and Rerun log sizes are comparable with a 256-memory-reference maximum episode size. Figure 10 displays Karma's uncompressed log size normalized to Rerun's uncompressed log size on a 16 core system. The x-axis varies both Karma and Rerun's maximum episode sizes (in memory references) for 128, 256, ..., 8K, and unbounded. For the default of 256 memory references, Karma's log sizes versus Rerun's varies from 10% smaller for Zeus to 17% larger for OLTP. As episodes grow larger, Karma's log size decreases faster than Rerun's. For maximum episode sizes of 1K references and greater, Karma's log is always smaller than Rerun's. With a maximum episode size of 8K references), Karma log size is smaller than Rerun's by 33%, 43%, 17% and 35% for Apache, Jbb, Oltp and Zeus, respectively.

If small logs are more important than faster replay, then it is reasonable to set the maximum episode size to 2K references. To this end we found out that with maximum episode size 2K references, Karma retains substantial replay speedups with respect to Rerun (e.g. 1.3X-4.8X) but they are smaller than with 256-reference episodes. Moreover, Figure 10 showed that Karma's log size is 8-35% smaller than Rerun's log size, when the maximum episode size is limited to 2K memory references in both systems.

7 Conclusions

This paper proposes *Karma* for both scalable recording and replay. Karma builds episode-based memory race recorder/replayer using a directory coherence protocol, without requiring any *global* communication. During recording, Karma records the order of episodes with a directed acyclic graph and extends episodes even after some conflicts. During replay, Karma uses wakeup messages to trigger parallel replay of independent episodes. Results with several commercial workloads on a 16-core multicore system show that Karma can achieve replay speed within 19%-28% of execution speed without record-replay and four times better than idealized Rerun replay.

8 Acknowledgements

We thank Derek Hower for providing the code for Rerun. We thank Brad Beckmann, Dan Gibson, Heidi Arbisi-Kelm, Rathijit Sen, Mike Swift, Haris Volos, Min Xu, the Wisconsin Multifacet group, the anonymous reviwers and the Wisconsin Computer Architecture Affiliates for their comments and/or proofreading. Finally we thank the Wisconsin Condor project, the UW CSL for their assistance.

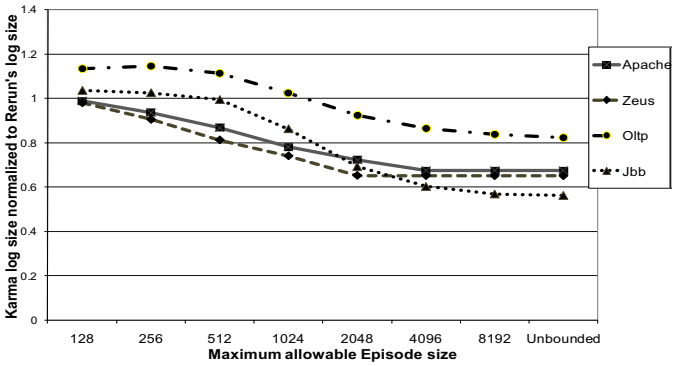

Figure 11. Log size comparison between Karma and Rerun

This work is supported in part by the National Sicence Foundation (CNS-0551401, CNS-0720565 and CNS-0916725), Sandia/DOE (#MSN123960/DOE890426), and University of Wisconsin (Kellett award to Hill). The views expressed herein are not necessarily those of the NSF, Sandia or DOE. Bobba was PhD student at University of Wisconsin-Madison when this work was performed. Hill has a significant financial interest in Microsoft.

References.

[1] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, February 2002.

[2] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, 2009.

[3] David F. Bacon and Seth Copen Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 194–206, 1991.

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proc. of the 15th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, March 2010.

[5] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[7] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[8] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 224–234, New York, NY, USA, 1991. ACM.

[9] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, March 2009.

[10] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of the 2002 Symp. on Operating Systems Design and Implementation*, pages 211–224, December 2002.

[11] George W. Dunlap, Dominic Lucchetti, Peter M. Chen, and Michael Fetterman. Execution Replay of Multiprocessor Virtual Machines. In *International Conference on Virtual Execution Environments (VEE)*, 2008.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March-April 2000.

[14] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manoit Juin-Yeu Joseph Lu, and Shridhar Narayanan. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

[15] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. Technical Report Technical Report 92/07, Digital Cambridge Research Lab, 1992.

[16] Mark D. Hill and Min Xu. Racey: A Stress Test for Deterministic Execution. http://www.cs.wisc.edu/ markhill/racey.html.

[17] Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Race Recording. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[18] Intel, editor. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A: System Programming Guide Part 1. Intel Corporation.

[19] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] Leslie Lamport. How to Make a Multiprocess Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, pages 690–691, 1979.

[21] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[22] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[23] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, and Peter Chen. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Proc. of the 15th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 77–90, March 2010.

[24] Dominic Lucchetti, Steven K. Reinhardt, and Peter M. Chen. ExtraVirt: Detecting and recovering from transient processor faults. In *2005 Symp. on Operating System Principles work-in-progress session*, October 2005.

[25] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[26] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, September 2005.

[27] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[28] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: A Software-Hardware Interface for Practical Determinisitic Multiprocessor Replay. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[29] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, pages 258–269, February 2006.

[30] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording Shared Memory Dependencies Using Strata. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, October 2006.

[31] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, pages 284–295, June 2005.

[32] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[33] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, October 2009.

[34] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern CMPs. In *Proc. of the 42nd Annual IEEE/ACM International Symp. on Microarchitecture*, pages 576–585, December 2009.

[35] Milos Prvulovic and Josep Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 110–121, June 2003.

[36] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *Proc. of the 41st Annual IEEE/ACM International Symp. on Microarchitecture*, November 2008.

[37] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd Annual Intnl. Symp. on Computer Architecture*, pages 4June 1995.

[38] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proc. of the 13th Annual Intnl. Symp. on Computer Architecture*, pages 414–423, June 1986.

[39] Gwendolyn Voskuilen, Faraz Ahmad, and T. N. Vijaykumar. Timetraveler: exploiting acyclic races for optimizing memory race recording. In *Proc. of the 37th Annual Intnl. Symp. on Computer Architecture*, pages 198–209, June 2010.

[40] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[41] Min Xu, Rastislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 122–133, June 2003.

[42] Min Xu, Rastislav Bodik, and Mark D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, October 2006.

[43] Min Xu, Vyacheslav Malyugin, Jeff Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, June 2007.

[44] Jie Yu and Satish Narayansamy. A Case for an interleaving constrained shared-memory multi-processor. In *Proc. of the 36th Annual Intnl. Symp. on Computer Architecture*, pages 325–336, June 2009.