Dynamic NUMA-Aware Data Structure Replication

Erika Hunhoff*‖, Zack McKevitt*, Ankit Bhardwaj†, Reto Achermann‡, Gerd Zellweger§‖
Marcos K. Aguilera¶‖, Eric Keller*
*University of Colorado at Boulder, USA, †Tufts University, USA, ‡Technical University of Munich, Germany, §Feldera,
¶Nvidia,‖Some work performed at VMware Research and/or Broadcom, Inc.
{erika.hunhoff, eric.keller}@{colorado.edu}

***Abstract*—Computing infrastructure is increasingly composed of non-uniform memory access (NUMA) architectures, such as machines with multiple sockets and rackscale systems with shared memory. A powerful technique to effectively use such architectures is to replicate data structures across NUMA domains, so threads can quickly access a local replica of data. Unfortunately, existing replication methods support only a static set of replicas, which is mismatched to the dynamic needs of modern systems. We introduce LD-NR, a replication technique that allows the number and location of replicas to change dynamically. LD-NR can adapt to workloads that change over time, such as applications that scale up and down across NUMA domains, whereas NR, the state of the art in NUMA-aware data structure replication, is limited to a static replica set. LD-NR achieves this benefit with negligible memory and performance overhead when configured identically to NR. We show that a hashmap dynamically replicated with LD-NR achieves 2.6 times better throughput than the best performing concurrent hashmap implementation, and provides 97% of the throughput provided by NR with an ideal, static configuration. We further demonstrate the utility of LD-NR to optimize complex systems by using it to enable a granular, per-process page table replication policy in a rackscale operating system.**

I. Introduction

Computing infrastructure is increasingly composed of non-uniform memory access (NUMA) architectures, whereby a processor can access locally attached memory faster than memory attached to other processors. Today, we find NUMA in the form of tiles [54] or chiplets within processors [3], [46], in the form of multi-socket servers, and even across servers in rackscale systems with shared memory [13], [16], [21], [55].

Despite the prevalence of NUMA, software designed for uniform memory access architectures can perform poorly on NUMA architectures due to the extra latency incurred on remote accesses: NUMA characteristics can impact application performance by up to 20% [57]. Many ideas have been proposed to address this issue, ranging from thread migration [31], [34], [43] to new NUMA-aware data structures and applications [14], [19], [29], [45], [47], [61], [62]. A general and powerful technique for adapting data structures to NUMA architectures is to replicate data across NUMA domains, so that a processor can access data locally on reads, while using a replication algorithm to update the replicas on writes [9].

Unfortunately, existing methods to replicate data across NUMA domains support only a *static* set of replicas—typically one per NUMA domain—which must be decided at initialization and subsequently cannot be changed. Static replication works well when the computational needs of the system are also static, but dynamic systems introduce several problems for static replication. First, applications and their workloads often vary over time. For instance, an application may use a few threads within a single NUMA domain during idle periods but use threads in many domains during peaks. For this application, replicating data across many domains is sometimes wasteful, but replicating data across few domains will not reach the full benefits of replication. Second, as the scale of applications and systems increases, it may be too expensive to naively replicate data across all domains. Third, new disaggregation technologies [16] allow the number of domains provided by the hardware to vary over time. For example, an operator may assign a new machine to a cluster on a rack in order to allow an application to scale up; in this case, expanding the replica set to include the new machine would be desirable. Fourth, the ideal number of replicas may fluctuate depending on the system-wide state dictated by resource scheduling algorithms. For example, if the system is running out of memory, it may be necessary to reduce the number of replicas to free memory.

We run several experiments to quantify the performance issues that arise when the system has the wrong number of replicas due to dynamic changes (§II-C). Although the memory overheads of replication are linear to the number of replicas, we find that the relative performance benefits across static replication configurations are heavily influenced by dynamic factors. We illustrate that it is not possible to optimize the memory-performance tradeoffs of replication in a dynamic system using a static policy. Furthermore, we identify cases where static replication can be detrimental beyond suboptimal performance. For instance, replication with specific workload characteristics can lead to stalls, where operations on the replicated data cannot complete.

To address these issues, we introduce a new method to replicate data across NUMA domains called Live-Dynamic Node Replication (LD-NR). LD-NR allows the number and location of replicas to change dynamically. This enables a user of LD-NR to adjust the performance-memory tradeoff of replication in response to system state. The three main challenges of dynamic replication are: (1) determining which replica to use for a given operation in the absence of a local replica, (2) ensuring liveness when some replicas are lagging behind, and (3) adaption of a replication algorithm to provision

978-1-xxxx-xxxx-x/xx/$31.00 © 20xx IEEE

or release replicas. LD-NR addresses these challenges as follows. First, it introduces a NUMA-aware dispatcher that matches threads to replicas while prioritizing locality (§III-C). Second, it provides a new mechanism for liveness whereby threads from other NUMA domains detect and update lagging replicas (§III-D). Third, it uses a new NUMA-aware replica cloning technique to adapt the replication factor (§III-B).

We evaluate LD-NR through comparison to concurrent data structure implementations and NR, a state-of-the-art NUMA-aware replication library that provides static replication only [9]. LD-NR maintains optimal replication configuration even in systems with dynamicity, and LD-NR incurs no memory overhead and nominal performance overhead when configured identically to NR: for a hashmap under contention, LD-NR achieves 2.6× the throughput over comparable concurrent hashmap implementations, and provides 97% of the throughput provided by optimally configured NR.

We further use LD-NR to implement DiNOS, an operating system (OS) for rackscale shared memory systems. The design of DiNOS is motivated by current resource disaggregation trends [13], [16], [21], [55] that allow a rack to have a variable number of compute domains sharing memory with each other. DiNOS uses LD-NR to replicate per-process page tables across the domains each process is currently executing. DiNOS reduces the memory overhead of replication (by pruning useless replicas) while replicating when it is useful (a process is active in a domain) and feasible (the system has enough memory). LD-NR's dynamic replication scheme has finer granularity than state-of-the-art page table replication [1], [42], [44].

Our contributions are as follows. First, we identify and quantify issues that arise from the static NUMA-aware replication of NR [9](§II). Second, we introduce a new NUMA-aware replication scheme called LD-NR that supports dynamic replication (§III). Third, we illustrate the power of LD-NR to optimize complex systems in emerging architectures by designing a new OS called DiNOS for rackscale environments (§IV). Fourth, we implement and evaluate LD-NR and DiNOS (§VI). Both LD-NR and DiNOS are open-source.

II. Background and Motivation

The primary method of optimizing NUMA systems is to maximize locality, with the accessor of memory ideally sharing the same NUMA domain as the memory being accessed. Intuitively, replication can increase NUMA locality, since all domains containing a replica of data are considered NUMA local. Given this, the Node Replication (NR) algorithm and library was designed to provide strongly consistent NUMA-aware replication for sequential data structures [9]. However, when properties of the system diverge from conditions assumed by NR, the benefits of NR dwindle while overheads remain. This is a problem, as some properties are shaped by dynamic behavior. Optimizing dynamic systems using NUMA-aware data structure replication requires *dynamic replication*.

Sec. II-A summarizes NR and provides an overview of the underlying assumptions and resulting performance characteristics. Sec. II-B highlights four system trends that illustrate the importance of dynamic replication as a tool for system optimization. Sec. II-C then empirically shows that NR's statically decided tradeoffs prevents system optimization.

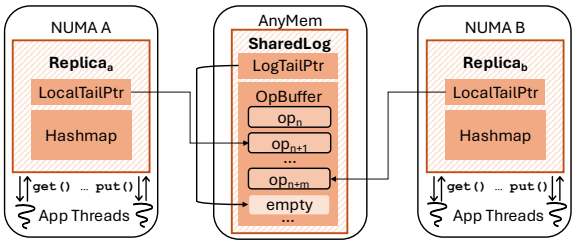


Fig. 1: A hashmap replicated by NR [9] across NUMA domains `A` and `B`. NR components are orange striped.

A. Background on Node Replication

NR provides NUMA-aware replication for arbitrary sequential data structures [9] using a shared log (Fig. 1 `SharedLog`) to which operations are appended to and consumed from (Fig. 1 `OpBuffer`). Each *operation* encodes an action that accesses or modifies the replicated data structure (Fig. 1 `get`, `put`). NR assumes a static assignment of application threads to cores (i.e., a thread is pinned to a core)—an assumption we will share. Each thread must register with a replica, and NR assumes threads register with a NUMA-local replica. Threads submit operations using the *execute* function provided by NR replicas. Each NR *replica* contains a copy of the data structure (Fig. 1 `hashmap`) and metadata (Fig. 1 `LocalTailPtrs`, which record the next operation in the shared log that the replica should consume).

The NR shared log, inspired by distributed systems such as CORFU [4], provides strong consistency through linearizability of operations across replicas. When a thread submits an operation, it attempts to become the per-replica *combiner* (an optimization technique called flat-combining [27]). There is at most one combiner per replica, and this combiner only exists in response to an application invoking `execute`. A replica is *inactive* when it is not being utilized, and thus there has been no combiner activity for some time.

The combiner for a replica handles operations submitted by multiple threads registered to the replica in batches. It *appends* operations that mutate the data structure to the log, ensuring that the operations will be applied to every copy of the data structure in the same order. The combiner of a replica *consumes* operations from the shared log by applying them to the replica's copy of the data structure. The result of an operation o can be computed once the combiner has consumed all operations submitted to the shared log before the submission of o; at this point, o can be applied safely and consistently to the local replica to obtain the result. As the shared log is stored in an unspecified NUMA domain (Fig. 1 `AnyMem`), actions on the shared log may require cross-NUMA memory accesses but operations on the local copy of the data structure do not. In addition to increasing NUMA-local accesses, NR also reduces contention since at most one

combiner per replica performs mutating operations on data structures, including the shared log.

B. Motivating Dynamic Replication

NR is designed for a static configuration of one replica per NUMA domain. We argue that the use of NR for system optimization requires dynamic configuration of the placement and number of replicas, breaking this core assumption. We outline four characteristics of modern systems that motivate the need for dynamic replication towards the goal of system optimization. This provides context for Section II-C, which empirically quantifies the extent that NR can be used for optimization under dynamic conditions.

Applications are dynamic: Workloads are variable [15], [49], [51]. Demands on applications change, leading applications to allocate or deallocate memory and spawn or release threads. Likewise, usage behavior changes between applications, deployment scenarios, and over time. This leads to variance in the access patterns, such as the read/write ratio and timing, of data structures. The set of replicas should be configured to meet dynamic application needs.

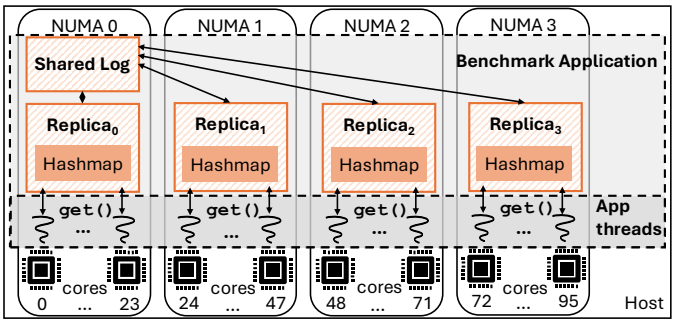
System scales are increasing: While hosts with one to four NUMA domains are common today, the scale and complexity of NUMA systems is increasing. This trend is driven by demand for scalable applications (e.g., machine learning) and supported by architectural advances (e.g., sub-NUMA domains [3], [46]). Memory disaggregation can also result in more complex NUMA topologies through dynamic cross-component scaling supported by high-performance interconnects [16], [28]. In large-scale systems, replication of every data structure on every NUMA domain incurs significant overhead; dynamic replication is needed to temper that overhead.

System resource scheduling is complex: Application performance and the benefits of disaggregation are diminished when applications are unable to effectively use the NUMA topology reflected in the resources assigned to the application. An application may not receive its ideal NUMA placement, particularly in a shared system. For instance, if one NUMA domain is short on memory, it may not be feasible to maintain a replica there. In a scenario with system memory pressure, a reduced replication factor may be necessary to avoid exceeding available memory. Dynamic replication is a key optimization tool for applications deployed in large-scale, shared systems.

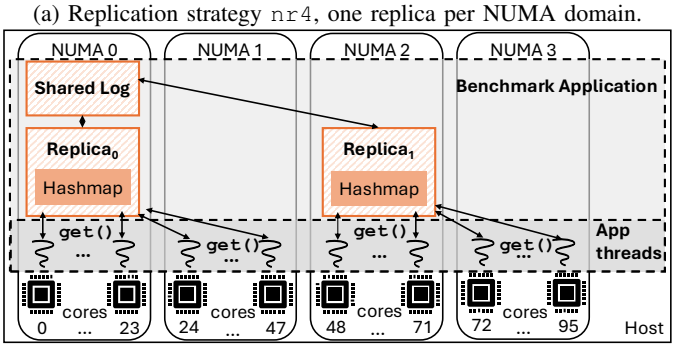
Systems are becoming dynamic: With disaggregation comes the ability to assemble a system runtime which requires optimizing the formation of a logical computer from a pool of disaggregated resources–a higher-level problem than our focus of system optimization. Nevertheless, dynamic replication is needed to enable an adaptable logical computer to utilize replication effectively.

C. Impact of NR's Static Replica Configuration

NR assumes applications statically allocate one replica per NUMA domain. Previous evaluations of NR also assumed a static workload. We quantify NR's sensitivity to dynamism by measuring NR's performance without these assumptions.



(a) Replication strategy `nr4`, one replica per NUMA domain.



(b) Replication strategy `nr2`, two replicas in two NUMA domains.

Fig. 2: Replication strategies for `NRHashmap` with 96 threads and a workload of all `get` operations. NR components are orange striped.

1) Experimental Setup: We created a benchmark application similar to related work [7], [9], [27] but modified to generate workloads that vary over time. We use the Rust implementation of NR [7] to replicate a hashmap, `NRHashmap`[1]. The test machine is a 96-core Intel Xeon Platinum 8260 host that has four equidistant NUMA domains of 24 cores each running Ubuntu 22.04.5 LTS. NUMA balancing and hyperthreading is disabled. Since the test machine has four NUMA domains, we test NR instantiated with one, two (Fig. 2b), and four replicas (Fig. 2a), respectively denoted `nr1`, `nr2`, and `nr4`. Application threads (hereby, threads) are balanced across replicas, and pinned to ensure locality between a thread and it's registered replica. If locality is not possible, threads are pinned to minimize the number of NUMA domains accessing each replica (shown for `nr2` in Fig. 2b).

Unless otherwise stated, each thread continuously applies `get` or `put` operations to the `NRHashmap`, with the choice of `get` or `put` chosen statistically according to a write ratio. The hashmap key for each operation is generated at uniform random. The hashmap is initialized with 67 million pre-allocated 64-bit keys and 64-bit values. Performance is aggregated per second across threads; memory utilization is measured using Rust allocator statistics [24].

2) Replica Count and Scale: We measured the effects of replication count and number of application threads on throughput (Fig. 3a) and memory utilization (Fig. 3d). At each

[1]A code snippet for this hashmap is included in a public artifact [59] and our implementation is open source.

stage in the experiment (delineated by gray blocks and labeled below the x-axis) the number of threads increases.

The best replication policy for throughput depends on the number of application threads with respect to the NUMA topology of the hardware. One replica (`nr1`) performs best when all threads t fit on one NUMA domain ($t \leq 24$). Once threads spill over to a second domain ($24 < t \leq 48$), `nr2` becomes the best policy. While `nr4` consistently uses the most memory, `nr4` only provides the best throughput for $48 < t \leq 96$. `nr1` and `nr2` show performance degradation once threads in different NUMA domains begin sharing a replica (> 24 and > 48, respectively). NR does not support changing the number of replicas outside of instantiation so it is not possible to pick a replication factor that provides the best throughput across all stages in this experiment.

3) Diminishing Returns on Memory Utilization for Given Workloads: Factors beyond scale influence the success of an NR configuration. Here, we consider three: write ratio (Fig. 3b), contention (Fig. 3e), and operation latency (Fig. 3c). All three experiments are configured with 96 threads. For write ratio in Fig. 3b, we increase the percentage of `put` to `get` operations in each stage. On average, `nr4` provides the best throughput; however, as write ratio increases, the gap between different replication strategies narrows. Figures 3e and 3c are each configured with 10% write ratio, and show a similar pattern. For contention (Fig. 3e), each thread executes `n` operations on a thread local hashmap (independent of the `NRHashmap`) between each `NRHashmap` operation. In Figure 3c, operation latency is artificially increased by modifying the hashmap wrapped by NR to perform `n` extra `gets` or `puts` each time the respective operation is called. The modification of the wrapped data structure is transparent to NR in both cases. It is natural for throughput to decrease both as we reduce contention and increase operation latency, as the highest theoretical throughput is also decreased.

These experiments illustrate that static replication can yield diminishing returns with equal memory overhead in dynamic systems. While memory overheads are marginal for small data structures replicated a few times, memory overhead scales linearly with data structure size and replica count.

4) Replica Inactivity and Stalling: Static registration of threads to replicas at initialization makes applications susceptible to NR stalls. NR can cause blocking if one or more replicas become inactive while other replicas remain active. Recall that a replica is active if threads that are registered to that replica are performing operations, since in the course of processing an operation the replica is updated. If a replica is inactive, space exhaustion on the shared log can cause aggregate throughput to drop to zero. While increasing the number of replicas potentially increases peak throughput, it also increases the possibility of an inactive replica.

Fig. 3f presents the effects of stalls. During each six-second interval, threads assigned to replica$_0$ are made idle for the middle two seconds, rendering replica$_0$ inactive. Threads assigned to other replicas continue to attempt to perform `get` and `put` operations using a 10% write ratio, gradually filling up the shared log with pending operations and eventually triggering a stall. Throughput falls to zero during stalls, regardless of the replication factor (`nr2` or `nr4` in Fig. 3f) and the total number of threads (48, 72, and 96 in Fig. 3f).

Existing techniques to mitigate stalls including a dedicated worker per replica [9] or a periodic timer per replica [5], [7]. However, there are downsides to both options. Dedicating resources can be wasteful when there is low utilization. Periodic timers still result in stalls, just stalls bounded by the timer interval; furthermore, threads interrupted by the timer are distracted from the work they would otherwise be doing. Additionally, both of these solutions rely on information about the number of replicas and which threads are assigned to each replica. This information is easy to track within the static assumptions of NR, but would be hard to track if those values became dynamic, as in dynamic replication.

5) Summary: Our evaluation of NR shows that dynamic factors influence the best replication policy for performance (Fig. 3a) as well the performance-memory tradeoff of replication configurations (Fig. 3b-3f). We also note that limitations of available memory can make certain replication configuration infeasible. Furthermore, we show that poor fits between workload characteristics and replication configurations are not only suboptimal, but can cause degradation (Fig. 3a) and stalls (Fig. 3f). Hence, we need a replication algorithm that can adapt to dynamic workload and system state.

III. Live-Dynamic Node Replication (LD-NR)

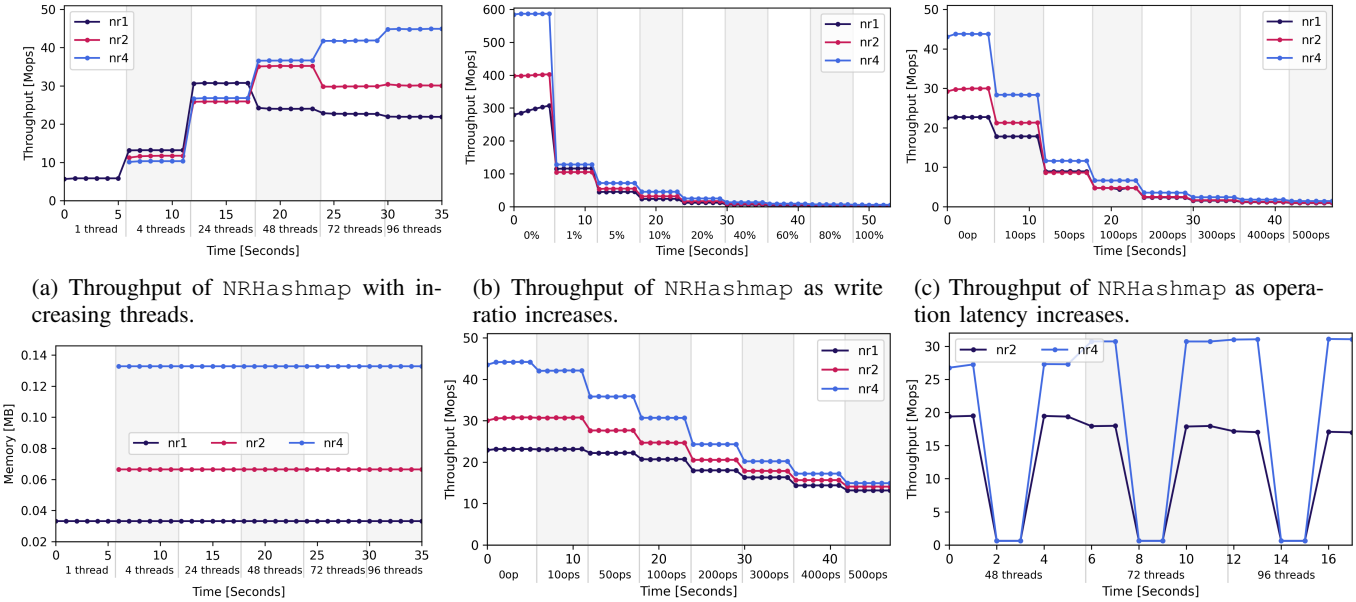
We introduce LD-NR, an algorithm that provides dynamic, NUMA-aware replication (§ III-A), and describe three new techniques used by LD-NR to support dynamic replication (§III-B–§III-D).

A. Design Overview

LD-NR is designed around two new components: a replica dispatcher and an affinity manager, illustrated in Fig. 4.

*a) The **replica dispatcher** serves two purposes:* it is a container for metadata and a gateway that application threads use to submit operations. LD-NR requires a container because metadata must be preserved independent of the existence of any replica. LD-NR requires a gateway because threads cannot assume the existence of any particular replica and thus need a static construct to mediate their access to replicas. The dispatcher preserves *liveness*, meaning threads can always access and perform operations on the replicated data structure.

When an application calls the dispatcher `execute` function, the dispatcher selects a replica to handle the operation. Thread-to-replica matching is transparent to the thread. The dispatcher also provides methods for adding and removing replicas. LD-NR does not change the replication factor unless the application directs it to do so. Exposing the mechanism enabling dynamic replication to the application rather than implementing specific dynamic replication policies internal to LD-NR allows LD-NR to be adapted to various data structures, workloads, and architectures.



(a) Throughput of `NRHashmap` with increasing threads.

(b) Throughput of `NRHashmap` as write ratio increases.

(c) Throughput of `NRHashmap` as operation latency increases.

(d) Memory utilization of `NRHashmap` with increasing threads.

(e) Throughput of `NRHashmap` as contention decreases.

(f) Throughput of `NRHashmap` with periods of replica inactivity.

Fig. 3: Performance in throughput (3a-3c,3e-3f) and memory (3d) of `NRHashmap`, across varied configurations. Each experiment phase runs for six seconds. In 3a and 3d, `nr2` and `nr4` start at four threads to ensure ≥ 1 thread per replica.

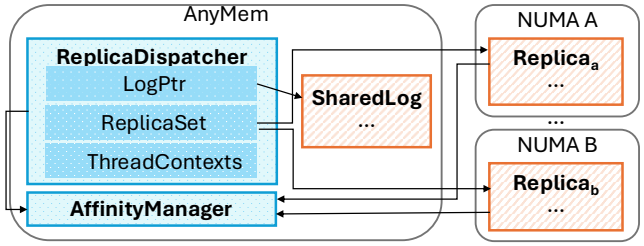


Fig. 4: Illustration of LD-NR. New components are dotted blue; components similar to NR are striped orange.

*b) The **affinity manager** has one purpose:* to modify/restore the memory allocation policy before/after actions that could allocate memory on behalf of a replica. A core concept of NUMA-aware replication is that each replica is constrained to a particular NUMA domain. In dynamic systems and where the dispatcher performs thread-to-replica matching, the affinity manager is invoked to set an affinity before an operation that might allocate memory executes (e.g., before a replica combiner applies operations) and then restore it afterwards. Previous replication techniques such as NR did not integrate a similar mechanism because in static systems this is often realized for free, e.g., if a thread only acts on a NUMA-local replica, default memory allocation policies such first-touch make it very likely that the replica will be allocated memory from the local NUMA domain. The additional capabilities of LD-NR, however, require an affinity manager to retain this property. Because LD-NR is a general-purpose library not tied to a particular OS, LD-NR requires the user to provide an implementation of an affinity manager. For instance, in Linux, we can implement an affinity manager using numactl [40]. Affinity managers are reusable, as they tend to use OS-specific (and not application-specific) capabilities. While the user provides an implementation, LD-NR internally invokes the manager as necessary, transparent to the user.

c) LD-NR includes two components modified from NR: replicas, and the shared log. These similarities allow LD-NR to serve a similar purpose to NR (replication of arbitrary sequential data structures) and inherit NR optimization techniques. However, these components required modifications. In LD-NR, the shared log is aware of the dynamic replica set, which is necessary to ensure garbage collection of the shared log occurs under the correct conditions. Also, replica-specific metadata must be globally unique in LD-NR in order to persist and be meaningful through replica set changes. LD-NR moved this information from replicas to the dispatcher, which required pervasive changes to how replicas access state.

B. NUMA- and Log-Aware Replica Cloning

When adding a new replica, LD-NR creates a copy of the replicated data structure. However, it is not sufficient to simply clone an existing replica as this cloning must be done in cooperation with the shared log for consistency, and memory must be allocated with the NUMA affinity of the new replica. We develop a NUMA- and log-aware replica cloning technique consisting of the following steps. The procedure for removing a replica follows a similar logic but loosely in reverse.

1. Select Model Replica: When an application initiates the addition of a replica, the dispatcher selects the replica that has

made the most progress (i.e., consumed the most operations from the shared log) as a model for the new replica, ensuring the new replica is not a straggler.

2. Freeze Replica Model: The dispatcher stops the model from performing any log operations (consume or append) while cloning is in progress. Freezing ensures the consistency of the data structure and its metadata, as replica cloning is not atomic. Freezing also stops operations from being garbage collected from the log before they are applied to the new replica. In the initial implementation of LD-NR, replica freezes are applied coarsely: the dispatcher freezes thread matching for all replicas. However, future work could allow unaffected replicas to continue unfrozen and even allow a partial freeze where the model replica is still able to apply appends.

3. Clone Model Replica: The dispatcher uses the affinity manager to change the memory allocation affinity of the application thread performing the dispatcher's `add_replica` operation to match the desired NUMA domain of the new replica. The dispatcher then copies the model replica, including both metadata and the data structure.

4. Update Metadata: The dispatcher adds the new replica to the replica set of the shared log and also adds the new replica to the replica set maintained by the dispatcher (enabling the dispatcher to match threads to the new replica).

5. Restore State: Lastly, the dispatcher restores the memory allocation affinity of the thread performing the add replica operation and unfreezes replicas. All replicas are now available.

C. Dynamic matching of threads to replicas

LD-NR requires dynamic matching of threads to replicas. The matching process must both ensure threads can access replicas under all conditions as well as ensure threads access a local replica (if one exists). LD-NR's dispatcher manages the set of available replicas and chooses a replica to handle the operation. The dispatcher selects the replica with the same NUMA affinity as the thread (which is recorded during registration), if one is available. Otherwise, the dispatcher selects the replica by load-balancing threads without local replicas across available replicas. Replicas must also keep track of the threads they are responsible for. The replica's combiner uses this information to gather in-flight operations from those threads for batching. In-flight operations from threads not acting as combiners must be handled by exactly one combiner, so it is essential that this process be accurate. The dispatcher is responsible for updating the set of threads each replica must manage; this update occurs when a replica is added or removed.

D. Avoiding Stalls with Affinity Management

In NR, a replica may become inactive if its threads do not execute operations, causing the pending operations in the shared log to build up until there is no free space. At that point, new operations cannot be added to the shared log, which causes active replicas to block. LD-NR uses the affinity manager, the metadata reorganization, and dynamic thread-to-replica matching to solve this problem. Recall that a thread

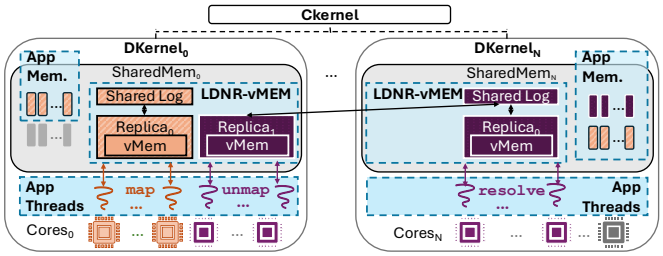


Fig. 5: LDNR-vMem replication for two processes on DiNOS.

becomes the per-replica combiner to process operations on the replica. In LD-NR, we use replica combiners to detect when they are unable to append operations to the shared log, the condition causing a stall. LD-NR fixes the stalling condition by allowing the combiner that detected the stall to attempt to also become the combiner of the replica that is farthest behind, thus becoming the combiner for two replicas. The attempt to become a second combiner may fail if another combiner is trying to do the same, but regardless, once there is a combiner working on the lagging replica, that replica is no longer inactive. Whichever thread succeeds in becoming a second combiner uses the affinity manager to match memory allocation affinity with the inactive replica, and then starts consuming operations from the shared log. This advances the state of the lagging replica, ensuring progress. The combiner repeats the process until all inactive replicas have advanced enough to allow garbage collection to free space in the operation log. This clears active replicas for progress.

IV. A Rackscale OS using LD-NR

We exercise LD-NR by building an OS for a rack with shared memory. We choose an OS as a complex application of LD-NR because OSes can control the scheduling affinities of threads accessing a replicated structure, which enables an OS to act as an ideal user of NUMA-aware replication. Reciprocally, since OS data structures can become bottlenecks to process performance, an OS can benefit from scalable, NUMA-aware data structures. OSes are expected to provide good performance over a range of dynamic scenarios, including scalable processes with varying workloads. We choose a rackscale OS because dynamic replication is especially important in the context of large or extended NUMA architectures (§ II-B). Section IV-A elaborates on the context of a rackscale OS and Section IV-B presents the OS design.

A. Context

Recent advances in optimized interconnects such as Compute eXpress Link (CXL) [16] can provide cache-coherent shared memory between interconnected hosts within a rack (∼2–16 hosts). Given the utility of such interconnects for disaggregated memory [32] and the trend towards scaling processes and VMs across hosts [10], [12], we posit these interconnects may enable new distributed OSes [2], [50].

We assume a rackscale OS is deployed over such a rack. Each host has a local shared memory region and can access the

shared memory regions of other hosts, but at higher latency. In addition to shared memory, hosts also contribute compute (their local cores) to the aggregate computational capacity of the rack. We assume the interconnect provides three features: first, cache coherence across hosts for shared memory regions; second, coherent shared memory can be accessed via load and store operations by each host; and third, interrupts can be sent between hosts over the interconnect.

B. Design

This section presents **D**ynam**i**c **N**r**OS** (DiNOS) design. While DiNOS is primarily designed to exercise LD-NR, resource management within the rack is a secondary priority. While OSes for NUMA [7], [8] and OSes for disaggregation [53] have been considered before, to our knowledge, the target scenario of DiNOS (flexible pooling of both memory and compute at rackscale supported by recent advances in interconnects) and focus (NUMA-aware replication) is novel. The design starting point for DiNOS is NrOS, a multikernel OS integrating NR [7]. In a DiNOS deployment, a NrOS-derived DiNOS component is deployed on each host in the rack. There are two primary components to DiNOS.

CKernel: One host is selected to run a controller instance (a CKernel) providing coarse-grained, global resource management. The CKernel schedules compute by the core and allocates memory by fixed-size, 2 GB, contiguous slices. Centralized resource management ensures the CKernel can use the holistic rack state for allocation and scheduling decisions [34]. Accordingly, the CKernel allocator and scheduler are designed in two levels to reflect a rackscale extended NUMA topology. The first level chooses the host within the rack; the second level chooses the particular resource within the host. The global allocator and scheduler 1) limits processes to the smallest amount of hosts, and 2) balances resources across the required amount of hosts. The CKernel also manages other control operations such as assignment of process identifiers.

DKernels: All other hosts run minimal data instances (DKernels), responsible for supporting processes using that DKernel's physical resources. DKernels forward resource requests and control operations to the CKernel via RPCs implemented over shared memory queues. Each DKernel contains LD-NR-replicated, per-process OS state for processes scheduled to use the DKernel's cores. DKernels also provide interrupt handlers.

1) LDNR-vMem: DiNOS manages per-process OS state across hosts using dynamic replication. Per-process OS state incudes the hardware page tables and page mappings that comprise the virtual address space of a process. This is analogous to NR-vMEM in NrOS [7], but when replicated with LD-NR, we refer to this state as LDNR-vMem. LDNR-vMem supports operations on a process' virtual address space such as `map` and `unmap`. These operations may be called explicitly by an application, such as when mapping a physical memory allocation, but often are called by userspace libraries for tasks such as resizing a process' virtual address space.

While NR-vMEM is statically replicated by NR across all NUMA domains for all processes, DiNOS instead uses LD-NR to enact a fine-grained, per-process, on-demand replication strategy to control the location and number of LDNR-vMem replicas. The LDNR-vMem for each process is only replicated on NUMA domains where it will provide increased locality for the process. Since LDNR-vMem replicas are only accessed in response to actions triggered by process threads of execution, the replicas must be placed only where the process is scheduled. For example (illustrated in Fig. 5), the orange striped process has one LDNR-vMem replica because it is scheduled on one DKernel's cores. Although this process is allocated memory on a second DKernel, there is no second replica. In contrast, the solid dark purple process scheduled on the cores of two DKernels has two LDNR-vMem replicas.

2) Cross-host Scaling: DiNOS allows a process to dynamically and transparently scale using any CPUs and any memory available in the rack irrespective of host boundaries with a unified process address space across hosts. This provides the benefits of disaggregation (e.g., reduced resource stranding [32], relaxation of bin packing [12]) but requires careful management of resources and OS state. As outlined above, DiNOS manages per-process OS state using LD-NR and centralizes resource management in the CKernel. In addition, DiNOS has an extended translation lookaside buffer (TLB) shootdown protocol that uses interrupts across hosts running a process. The CKernel uses a global resource accounting scheme that provides rack-unique core identifiers and rack-unique shared memory region affinity identifiers. While these considerations were required to make DiNOS function, these techniques are not novel so we do not describe them here. Known techniques including coarse-grained resource allocations, multi-level allocation logic, allocation-free shared memory RPCs, page caches on DKernels, and more are used to reduce the incidence of CKernel bottlenecks.

V. Implementation

LD-NR is implemented in Rust in approximately 1800 lines of code (LoC) with an additional 1200 LoC of tests. While shared logic exists between NR and LD-NR, most LoC were modified due to changes in types (global instead of replica-local thread identifiers, etc.) and changes to external and internal APIs. DiNOS is implemented in Rust using NrOS [7] as a base. As DiNOS is presented to illustrate the utility of LD-NR, we omit the discussion of several features of DiNOS in this paper. As such, it is difficult to get an accurate LoC count for the presented version of DiNOS but it easily exceeds 5k LoC. While theoretically capable of running unmodified POSIX binaries, currently DiNOS provides a limited, hypervisor-like API inherited from NrOS [7] that supports native DiNOS processes and a limited set of POSIX processes packaged within the NetBSD rumprun unikernel [39], [48]. DiNOS is deployable via emulation using KVM [25] and QEMU [6], with QEMU inter-VM shared mememory (ivshmem) [18] regions simulating cache coherent shared memory regions. Because the test machine has limited support for sub-NUMA

domains, in the prototype of DiNOS, the second-level memory allocator in the CKernel does not consider host-level NUMA domains.

VI. Evaluation

We evaluate LD-NR in dynamic situations in two different use cases: a hashmap (§VI-A), and per-process OS state in DiNOS (§VI-B). All experiments used the test machine described in §II-C1. We answer the following questions. First, can LD-NR provide the benefits of NUMA-aware replication under dynamic conditions? Second, does LD-NR incur minimal overheads compared to an optimally configured static replication scheme? Third, does NUMA-aware dynamic replication benefit data structures and the applications that use them?

A. LD-NR Hashmap

To answer the first evaluation question, we test the performance of an LD-NR-replicated hashmap under three varying factors: replication count (§VI-A1), both scale and replication count (§VI-A2), and replica inactivity (§VI-A3). To address the second question, we compare the LD-NR-replicated hashmap to an NR-replicated hashmap with static configurations of 1, 2, or 4 replicas (`nr1`, `nr2`, and `nr4` from §II-C1). To answer the third question, we compare the LD-NR-replicated hashmap against additional concurrent hashmap implementations (§VI-A2).

The code to wrap the hashmap for replication and define the encodings of `get` and `put` operations is nearly identical between LD-NR and NR. However, LD-NR additionally requires an affinity manager, which we implement in ∼30 LoC using a system call to `mbind` to set and restore the NUMA memory allocation policy for a thread. Identical underlying hashmap implementations are used; this comparison allows us to measure overheads incurred by LD-NR. If NR is optimally configured, LD-NR performance should similar to NR; if NR is not optimally configured, LD-NR should provide better performance.

1) Performance-Memory Tradeoff: We first demonstrate whether LD-NR can explore the performance-memory tradeoff of replication by varying the replication count.

Methodology. Keeping other factors steady (96 cores, 10% write ratio), we modify the replication factor over time for LD-NR by removing replicas (at 5.5, 11.5, and 17.5 seconds) and adding replicas (at 23.5, 29.5, and 35.5 seconds).

Analysis. The staggered pattern in Fig. 6a shows that throughput and memory use change corresponding to the replication factor. LD-NR allows a user to navigate the trade-off of using more memory to achieve better throughput.

2) Scaling with Dynamic Replication: We now evaluate whether LD-NR can be effectively used to enact a dynamic replication policy in response to a dynamic workload. As noted in Section II-C2, no single static NR policy provided the best throughput for `NRHashmap` across application thread counts. Since applications threads generate work in a closed loop as fast as possible, changes in thread count change the load on the data structure (e.g., the overall rate of requests), constituting a dynamic workload. We construct a dynamic policy for LD-NR that adopts the best NR configuration at each phase.

Methodology. We measure throughput and memory usage while scaling up the number of threads. We compare against NR and other concurrent hashmaps implemented in Rust, including: the standard collection HashMap in Rust (std) [58], an implementation that uses multiple locks over buckets to avoid global contention (CHashMap) [11], a port of Java's concurrent HashMap to Rust (flurry) [22], the User-space Read-Copy-Update (urcu) library's lock-freenhash table [36], [37], [52], and DashMap [63].

Analysis. Shown in Fig. 6b, LD-NR is competitive with the best static NR policy at each phase. Using the middle two seconds of each phase, LD-NR throughput is between 74.7% (phase 0, 1 thread) and >100% (phase 3, 48 threads) of the best NR throughput measured for that phase. LD-NR also uses memory conservatively in the early phases of the experiment, especially compared to `nr4`. LD-NR incurs a small overhead compared to NR when configured identically due to the additional checks and logic to handle dynamic configuration changes (e.g., at 96 threads with four replicas each, LD-NR sees 97.6% of the throughput of NR). Compared to other hashmaps in Fig. 6c, LD-NR does not provide the best performance with small thread counts (<24), but scales with significantly higher throughput than other concurrent hashmaps (2.6× better than urcu, the hashmap with the second best perfomrance in Fig. 6c). This underscores the importance of NUMA-aware replication as an optimization technique.

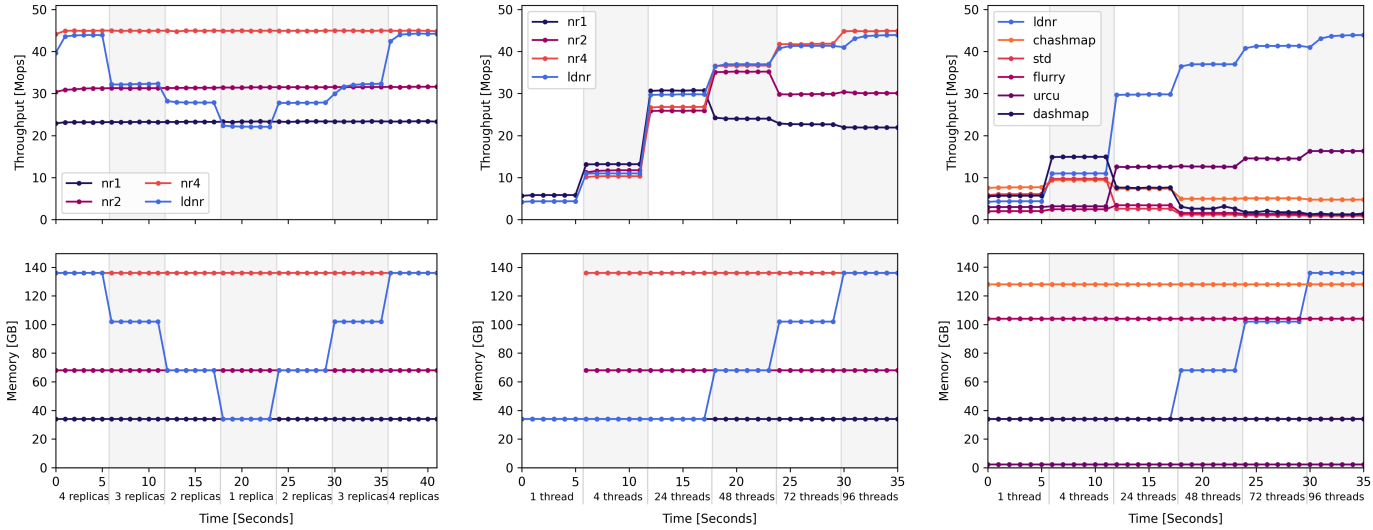
3) Tolerance for Inactive Replicas: LD-NR improves liveness by overcoming stalls when some replicas are inactive (e.g., when their threads do not execute data structure operations). We gauge the effectiveness of this technique by measuring throughput during intentionally triggered stalls.

Methodology. We use the experimental setup from §II-C4. Recall that the number of threads changes in each six-second phase and during the middle two seconds of each phase, the threads assigned to replica_0 do nothing, rendering replica_0 inactive and triggering a stall. LD-NR is configured to adapt the number of replicas to the number of active threads in each phase: two replicas for 48 threads, three replicas for 72 threads, and four replicas for 96 threads.

Analysis. Shown in Fig. 7, LD-NR provides liveness even with an inactive replica. At higher scale (96 threads) there is a performance drop, which is natural given that one of the combiners must perform twice the work during the period of inactivity and because fewer application threads are active.

B. DiNOS and LD-NR

We evaluate LD-NR in a more complex setting, DiNOS. We demonstrate that DiNOS is capable of supporting dynamic cross-host scaling of processes (§VI-B1) and use LD-NR to navigate the performance-memory tradeoff of replication (§VI-B2). In both cases, we select memcached, a common in-memory key-value store, as the process. We measure the throughput of memcached for random `get` requests on a 64 GiB dataset of 8-byte keys and 64-byte values. Note that



(a) Changing replica count to navigate the performance-memory tradeoff of replication.

(b) Scaling up threads for LD-NR compared to NR with static replicas.

(c) Scaling up threads for LD-NR compared to other Rust hashmap implementations.

Fig. 6: A suite of benchmarks showing the performance of `LDNRHashmap` when the replication policy changes over time (6a) and when both scale and replication policy change over time (6b, 6c).

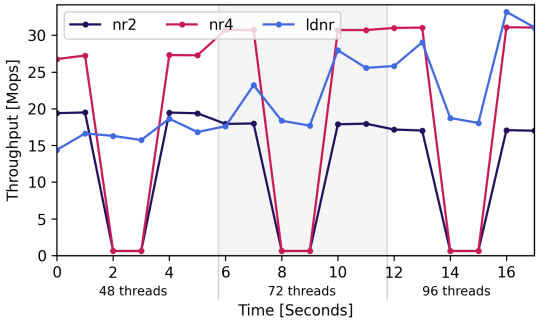


Fig. 7: When NR encounters stalls, LD-NR continues to provide throughput without application intervention.

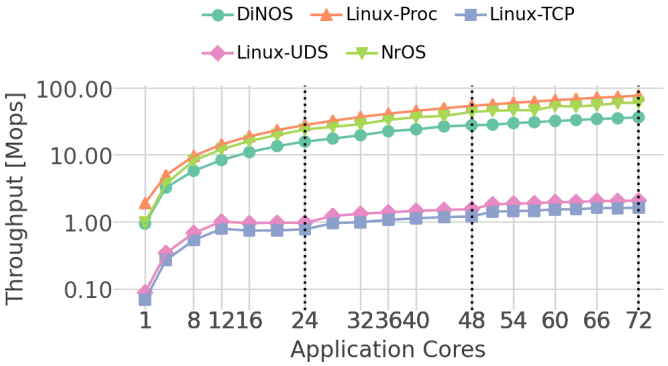


Fig. 8: Throughput of memcached `get` operations for scale up (DiNOS, Linux-Proc, NrOS) and scale out (Linux-UDS, Linux-TCP) configurations.

memcached `get` operations are not operations submitted to LD-NR, but are operations specific to memcached.

Setup. Each DiNOS component runs as a QEMU guest on a Linux host with resources from a single host NUMA domain (memory, shared memory, and cores). By running guests on different NUMA nodes, we create a performance difference between local and remote shared memroy regions. The test machine has four NUMA domains so we configure experiments with one CKernel and 1–3 DKernels.

1) Cross-Host Scaling: When a process needs to scale, it has two primary options: to *scale up* by using more resources on a single host or to *scale out* as a distributed process across hosts. One motivation for designing a rackscale OS is to support processes larger than a single host through cross-host scaling while avoiding common overheads of scale out. This experiment evaluates whether DiNOS achieves efficient scale up performance while transparently using resources across hosts.

Methodology. We configure memcached as a single process (scale up) or as 1–3 statically sharded processes (scale out). LinuxProc, NrOS, and DiNOS are scale up memcached deployments. DiNOS is a special case since host boundaries exist but are transparent to memcached. Linux-TCP and Linux-UDS are scale out deployments which use a gateway process to distribute the workload across memcached shard processes. TCP and Unix domain sockets (UDS), respectively, are how the gateway process communicates with the memcached shards. Fig. 8 shows the throughput (y-axis, millions of operations per second) for different configurations (x-axis, total application cores used by memcached). We use statically configured memcached executions (each point is a separate run) because the NrOS baseline cannot dynamically change the replica set. We sized NrOS to have the best performance for a given

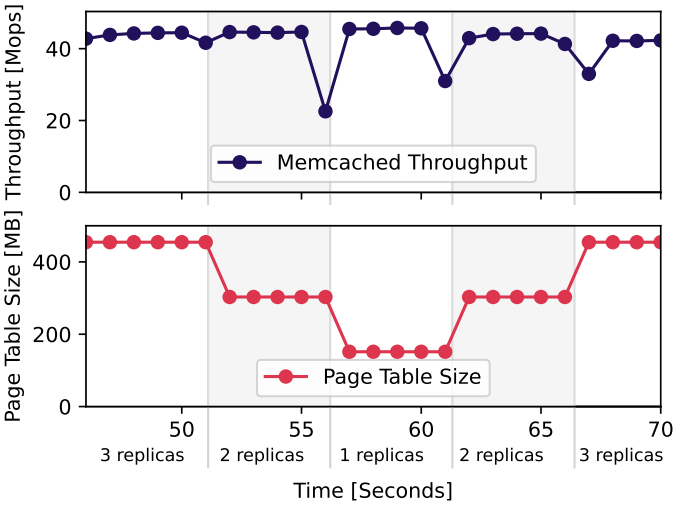


Fig. 9: Throughput of memcached `get` operations (top) and memory utilization of the replicated page table (bottom) as DiNOS changes LD-NR-vMEM replication factor.

application size. Dotted lines indicate when an additional DKernel (or memcached shard) is added (one shard/DKernel for $1 \leq t < 24$, two for $24 \leq t < 48$ threads, etc.).

Analysis. The single process and single host cases (Linux-Proc, NrOS) illustrate the upper-bound of performance. DiNOS shows comparable performance with these configurations, indicating that DiNOS successfully hides the characteristics of the emulated rack. In contrast, the scale out configurations exhibit performance an order of magnitude lower due to the additional communication required between the gateway process and the memcached shard processes. These results emphasize the motivation for a rackscale OS.

2) LD-NR in LDNR-vMem: DiNOS uses LD-NR to replicate process virtual address spaces (vMEM) including the page tables and meta data. We show that LD-NR enables the DiNOS to navigate the performance-memory tradeoff.

Methodology. We use a static configuration of DiNOS (3 DKernels with 24 cores each) and memcached (72 threads, 64 GiB dataset). While memcached is running, at specific intervals, we dynamically adjust the replication factor of memcached's LD-NR-vMEM by dropping then re-adding replicas. Fig. 9 (bottom) shows the memory requirements of the per-process vMEM supporting the virtual address space of memcached, while Fig. 9 (top) shows memcached throughput. We show results starting at 45 seconds, to allow memcached throughput to stabilize before we modify the replication factor.

Analysis Memcached's throughput is largely stable, with temporary drops when we adjust the number of replicas due to replica freezes (described in §III-B). Once the replica has been added or removed throughput returns to the stable state even when the replica set is reduced; this is because memcached's vMEM structure is not under enough load to benefit from having three replicas. However, dropping replicas greatly decreases the amount of memory required as memcached's vMEM structure is large due to memcached's virtual address space size. This shows that an OS can use LD-NR's granular replication controls to navigate performance-memory tradeoffs.

VII. Related Work

Data Structures and NUMA Many works adapt specific data structures to NUMA architectures, e.g., [29], [45], [62]. Other works build entire NUMA-optimized applications designed for tasks such as graph analytics and queries, e.g., [41], [64]. In contrast, LD-NR provides one piece of the optimization equation for an application: the method of NUMA-aware replication for data structures. Similar in focus, Shaol [30] replicates read-only data in array-based structures according to static data access patterns. In contrast, LD-NR supports sequential data structures, modifying operations, and allows a user to enact dynamic replication policies. LD-NR shares the general-purpose library structure of NR [9] but NR only supports static replication policy. While NR has previously been extended (e.g., CNR [7], verification [26]), LD-NR is the first to provide dynamic replication.

NUMA Architectures Initial works on NUMA often targeted servers with multiple sockets, but recent works target additional architectures, such as CXL-based architectures [32], [56] and processors with sub-NUMA (tiled) characteristics [35], [46], [54]. SWARM [38] explores replication protocols specialized for disaggregated memory accessed without cache coherence. LD-NR, and DiNOS, assume cache coherence; the problems faced in the two different scenarios result in solutions with fundamentally different structure. However, future research could include redesigning the shared log in LD-NR to function without cache coherence.

OSes, NUMA, and Replication The synergy between OS design, NUMA-awareness, and replication is clear. In Linux, AutoNUMA and extensions automatically optimize the NUMA-locality of processes based on dynamic factors [23], [34]. Locus [60] and Harp [33] replicate files for availability and performance. Tornado [20] has a notion of clustered objects supporting partitioning and replication, with local representations created upon first access.

There has been a specific focus on page table replication, due to its importance for process performance. Carrefour [17] automatically replicates user pages in the kernel. Mitosis [1] and vMitosis [42] replicate page tables on NUMA systems, with a focus on factors impacting the per-process decision to replicate or not, and provide a replication mechanism tailored for page tables. Similarly, Wasp [44] provides automated policies for enabling or disabling replication of page tables. NrOS [7] uses replication on all NUMA nodes for process page tables. In contrast to LD-NR, none of these systems allow for dynamically changing the number of replicas beyond full replication on all NUMA nodes or even processors, and no replication at all. Although DiNOS uses LD-NR to manage page table replication, LD-NR is not OS- or page table-specific.

VIII. Conclusion

We introduce a new method of replicating data structures across NUMA domains called Live-Dynamic Node Replication (LD-NR). LD-NR addresses a key limitation of prior work—a static inflexible set of replicas—to address the dynamic needs of modern systems (§II). LD-NR allows a user to adjust the number and location of replicas dynamically in response to changes in workload and system characteristics. We show that LD-NR improves performance of general data structures, and we use LD-NR to build a rackscale OS called DiNOS that replicates internal data structures.

In future work, LD-NR could be used as a basis to develop new policies for replication in other contexts; LD-NR could also be further extended to disaggregated settings to provide fault tolerance for replicas and the shared log.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. Memory disaggregation: why now and what are the challenges. *SIGOPS Oper. Syst. Rev.*, 57(1):38–46, June 2023.

[3] AMD. Socket SP3 platform NUMA topology for AMD family 19h models 00h–0fh. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/design-guides/56795_1_00-PUB.pdf, 2021. Accessed: 2025-03-30.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. USENIX, April 2012.

[5] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Log-structured protocols in Delos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 538–552, October 2021.

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, USA, 2005. USENIX Association.

[7] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective replication and sharing in an operating system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 295–312. USENIX Association, July 2021.

[8] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[9] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. FluidMem: Full, flexible, and fast memory disaggregation for the cloud. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 665–677, 2020.

[11] CHashMap: Fast, concurrent hash maps with extensive API. https://crates.io/crates/chashmap, 2019. Accessed: 2025-04-08.

[12] Ho-Ren Chuang, Karim Manaouil, Tong Xing, Antonio Barbalace, Pierre Olivier, Balvansh Heerekar, and Binoy Ravindran. Aggregate VM: Why reduce or evict VM's resources when you can borrow them from other nodes? In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 469–487, New York, NY, USA, 2023. Association for Computing Machinery.

[13] CCIX Consortium. An introduction to CCIX. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf, 2019. Accessed: 2025-04-17.

[14] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. DiffLex: A high-performance, memory-efficient and NUMA-aware learned index using differentiated management. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 62–71, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff, Prerit Oberai, Eric Keller, Eric Rozner, and Richard Han. Escra: Event-driven, sub-second container resource allocation. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 313–324, 2022.

[16] CXL. About CXL. https://computeexpresslink.org/about-cxl/, 2025. Accessed: 2025-03-30.

[17] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery.

[18] The QEMU Project Developers. Inter-VM shared memory device. https://www.qemu.org/docs/master/system/devices/ivshmem.html, 2025. Accessed: 2025-04-08.

[19] Dave Dice and Alex Kogan. Compact numa-aware locks. EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 87–100, USA, 1999. USENIX Association.

[21] Gen-Z consortium. https://genzconsortium.org.

[22] Jon Gjengset. flurry: Rust port of Java's ConcurrentHashMap. https://crates.io/crates/flurry, 2024. Accessed: 2025-04-08.

[23] Mel Gorman. Foundation for automatic NUMA balancing. https://lwn.net/Articles/523065/, 11 2012. Accessed 2025-04-16.

[24] Marcus Griep. stats_alloc: An allocator wrapper that allows for instrumenting global allocators. https://crates.io/crates/stats_alloc, 2025. Version 0.1.10. Accessed: 2025-03-30.

[25] Irfan Habib. Virtualization with KVM. *Linux J.*, 2008(166), February 2008.

[26] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 911–929, Boston, MA, July 2023. USENIX Association.

[27] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[28] Intel. An introduction to the Intel QuickPath interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009. Accessed: 2025-03-30.

[29] Safdar Jamil, Abdul Salam, Awais Khan, Bernd Burgstaller, Sung-Soon Park, and Youngjae Kim. Scalable NUMA-aware persistent b+-tree for non-volatile memory devices. *Cluster Computing*, 26(5):2865–2881, November 2022.

[30] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: smart allocation and replication of memory for parallel programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 263–276, USA, 2015. USENIX Association.

[31] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on NUMA systems: asymmetry matters. In *Pro-*

ceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '15, page 277–289, USA, 2015. USENIX Association.

[32] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 226–238, New York, NY, USA, 1991. Association for Computing Machinery.

[34] Jianguo Liu and Zhibin Yu. Global-state aware automatic NUMA balancing. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, Internetware '24, page 317–326, New York, NY, USA, 2024. Association for Computing Machinery.

[35] Ye Liu, Shinpei Kato, and Masato Edahiro. Analysis of memory system of tiled many-core processors. *IEEE Access*, 7:18964–18977, 2019.

[36] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-protected hash tables. https://lwn.net/Articles/573431/, November 2013. Accessed: 2025-04-08.

[37] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery.

[38] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. SWARM: Replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 24–45, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A dynamically adaptable multiserver-library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, page 114–128, March 2020.

[40] numactl(8) - Linux man page. https://linux.die.net/man/8/numactl, 2012. Accessed 2025-04-17.

[41] Peitian Pan, Chao Li, and Minyi Guo. CongraPlus: Towards efficient processing of concurrent graph queries on NUMA machines. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):1990–2002, 2019.

[42] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized NUMA servers with vmitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 194–210, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Panayiotis Petrides and Pedro Trancoso. Heterogeneous- and NUMA-aware scheduling for many-core architectures. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Hongliang Qu and Zhibin Yu. WASP: Workload-aware self-replicating page-tables for NUMA servers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1233–1249, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Maryan Rab, Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. NUMA-aware non-blocking calendar queue. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–9, 2020.

[46] TIRIAS Research. AMD optimizes EPYC memory with NUMA. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/AMD-Optimizes-EPYC-Memory-With-NUMA.pdf, 2018. Accessed: 2025-03-30.

[47] Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, Abhinav Vishnu, Dipanjan Sengupta, and Xu Liu. NUMA-caffe: NUMA-aware deep learning neural networks. *ACM Trans. Archit. Code Optim.*, 15(2), June 2018.

[48] Rumprun. https://github.com/rumpkernel/rumprun. Accessed: 2025-04-08.

[49] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[52] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[54] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[55] Jeffrey Stuecheli, William J Starke, John D Irish, L Baba Arimilli, D Dreps, Bart Blaner, Curt Wollbrink, and Brian Allison. IBM POWER9 opens up a new era of acceleration enablement: Opencapi. *IBM Journal of Research and Development*, 62(4/5):8–1, 2018.

[56] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL memory with genuine CXL-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–197, 2013.

[58] The Rust Programming Language. std::collections. https://doc.rust-lang.org/std/collections/, 2024. Accessed: 2025-04-08.

[59] VMWare, Inc. node-replication. https://github.com/vmware/node-replication, 2019. Accessed: 2025-04-10.

[60] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, page 49–70, New York, NY, USA, 1983. Association for Computing Machinery.

[61] Qing Wang, Youyou Lu, Junru Li, Minhui Xie, and Jiwu Shu. Nap: Persistent memory indexes for NUMA architectures. *ACM Trans. Storage*, 18(1), January 2022.

[62] Zhonghua Wang, Kai Lu, Jiguang Wan, Hong Jiang, Zeyang Zhao, Peng Xu, Biliang Lai, Guokuan Li, and Changsheng Xie. NStore: A high-performance NUMA-aware key-value store for hybrid memory. *IEEE Transactions on Computers*, 74(3):929–943, 2025.

[63] Joel Wejdenstål. DashMap: Blazing fast concurrent HashMap for Rust. https://crates.io/crates/dashmap, 2024. Accessed: 2025-04-08.

[64] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193, January 2015.

Artifact Description (AD)

IX. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The primary contributions of this paper are LD-NR and the frameworks and applications used for benchmarking LD-NR:

- C_1 LD-NR a library that supports dynamic replication
- C_2 Benchmarking suite for comparing LD-NR to NR
- C_3 DiNOS: An operating system with LD-NR integration

B. Computational Artifacts

- A_1 https://doi.org/10.5281/zenodo.18700666, also available at https://github.com/gz/node-replication
- A_2 https://doi.org/10.5281/zenodo.18700387, also available at https://github.com/hunhoffe/nrk

Artifact ID	Contributions	Paper Elements
A_1	C_1, C_2	Figure 3, 6, 7
A_2	C_3	Figures 8, 9

X. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact contains the implementation of LD-NR (C_1), the implementation of baseline NR, and a benchmarking framework for comparing the two against each other and against other Rust hashmap implementations (C_2). Specifically, the branches used to generate data for this work are:

- `nr-dynrep-microbench`: LD-NR implementation and configuration for hashmap benchmarks.
- `baseline-microbench`: Suitable NR configuration for apples-to-apples comparison to LD-NR.
- `dynrep-dinos`: Configuration for integration with DiNOS (A_2).

Expected Results

We expect the behavior of baseline NR vs LD-NR to behave similarly to the data presented in Figures 3, 6, and 7.

Expected Reproduction Time (in Minutes)

It may take several hours to run these experiments, depending on the computational capabilities of the machine.

Artifact Setup (incl. Inputs)

Hardware: 96-core Intel Xeon Platinum 8260 host that has four equidistant NUMA domains of 24 cores each running Ubuntu 22.04.5 LTS.

Software: The requirements are recorded in the artifact, using `cargo.toml` files, and further instructions are found in this appendix under the example configuration.

Datasets / Inputs: No data sets required.

Installation and Deployment: While the final results were not run on CloudLab nodes, an example setup on CloudLab follows:

- Image: Ubuntu 22.04
- Node type: d430
- Clone GitHub repo

```
git clone \
  git@github.com:gz/node-replication.git \
  -b nr-dyn-rep-microbench
```

- Run installations/updates:

```
sudo apt update
sudo apt install libhwloc-dev gnuplot \
  libfuse-dev liburcu-dev pkg-config --yes
sudo apt-get -y install libclang-dev
```

- Install rustup

```
curl https://sh.rustup.rs -sSf \
  | sh -s -- -y
source "$HOME/.cargo/env"
rustup update
```

- Build

```
cd node-replication/node-replication
cargo build
```

- Execute tests using Cargo

```
cargo test -p nr2
```

Note that the benchmarks for LD-NR measure short operations and so are sensitive to system configuration. Some commands useful for configuration follow:

```
# To set to performance
sudo cpupower frequency-set -g performance

# To set the highest available
# frequencies as the maximum
sudo cpupower frequency-set -u 3.90GHz

# To disable numa balancing
echo 0 | sudo tee \
    /proc/sys/kernel/numa_balancing

# To disable hyperthreading
echo off | sudo tee \
    /sys/devices/system/cpu/smt/control
```

Artifact Execution

The tests are orchestrated using typical cargo testing utilities. An example for how to run one of the subgraphs in Figure 6 is:

```
cargo bench \
  --features nr,thread-wait,wait-sweep \
  --bench hashmap
```

Other features include `cmp` (to compare against other hashmap implementations), `liveness`, and `threads`; there is roughly one feature that corresponds to each subgraph.

Artifact Analysis (incl. Outputs)

The output of the cargo tests are CSV files containing experimental data.

B. Computational Artifact A_2

Relation To Contributions

This artifact contains the implementation of DiNOS (C_3), which includes integration of A_1 as a git submodule. This artifact also contains a working version of NrOS for use as a baseline.

Expected Results

We expect the behavior DiNOS to behave similarly to the data presented in Figures 7 and 8.

Expected Reproduction Time (in Minutes)

The data for Figure 8 took 1 day to generate. The data for Figure 9 took approximately 3 hours to generate. The large delay in these cases is largely due to memcached pre-populating itself with a large amount of data.

Artifact Setup (incl. Inputs)

Hardware: 96-core Intel Xeon Platinum 8260 host that has four equidistant NUMA domains of 24 cores each running Ubuntu 22.04.5 LTS. Note that DiNOS is quite sensitive to different hardware, firmware, and qemu versions. As of this writing, none of the four socket machines available through CloudLab meet these requirements.

Software: The requirements are documented in the installation instructions for NrOS, which are publicly available, e.g., https://nrkernel.systems/book/environment/Environment.html. Note that the custom install of qemu is necessary to run the benchmarks. In general, rust requirements and versions are recorded in `cargo.toml` files within the artifact.

Datasets / Inputs: No data sets required.

Installation and Deployment: To generate the results in this paper, hugepages must be configured. A sample configuration is:

```
sudo apt install \
  -y libhugetlbfs-dev libhugetlbfs-bin
# Note: assume hugepagesize is 2048 kB
echo 131072 | sudo numactl -m 0 \
  tee -a /proc/sys/vm/nr_hugepages_mempolicy
echo 262144 | sudo numactl -m 1 \
  tee -a /proc/sys/vm/nr_hugepages_mempolicy
echo 393216 | sudo numactl -m 2 \
  tee -a /proc/sys/vm/nr_hugepages_mempolicy
echo 524288 | sudo numactl -m 3 \
  tee -a /proc/sys/vm/nr_hugepages_mempolicy
```

Artifact Execution

Our benchmarks – and the general unit and integration tests for NrOS and DiNOS– are configured as cargo tests. The data for Figure 8 is generated through two commands:

```
RUST_TEST_THREADS=1 cargo test \
  --features baseline,affinity-shmem \
  --test 's11*' \
  -- <testname> \
  --nocapture
```

where the testname is s11_rackscale_shmem-_memcached_internal_benchmark.

This will generate the data from running NrOS and DiNOS. To generate the data for Linux TCP and Linux UDP, run:

```
RUST_TEST_THREADS=1 cargo test \
  --test 's11*' \
  -- --exact <testname> \
  --nocapture
```

where the testname is s11_linux_memcached_sharded-_benchmark.

Similarly, the data in Figure 9 can be generated using the `dynrep` test using cargo. The throughput is output through a CSV, as in other tests. However, in contrast to other tests, statistics on memory utilization must be parsed from the test output (the output includes the size of the page tables/replicas throughout the test).

Artifact Analysis (incl. Outputs)

The output of the cargo tests are CSV files containing experimental data, as well as the stdout output for the dynrep test.