ConCORD: Easily Exploiting Memory Content Redundancy Through the Content-aware Service Command

Lei Xia
VMware
leix@vmware.com

Kyle Hale
Department of Electrical Engineering and Computer Science
Northwestern University
kh@northwestern.edu

Peter Dinda
Department of Electrical Engineering and Computer Science
Northwestern University
pdinda@northwestern.edu

ABSTRACT

We argue that memory content-tracking across the nodes of a parallel machine should be factored into a distinct platform service on top of which application services can be built. ConCORD is a proof-of-concept system that we have developed and evaluated to test this claim. Our core insight is that many application services can be described as a query over memory content. This insight leads to a core concept in ConCORD, the content-aware service command architecture, in which an application service is implemented as a parametrization of a single general query that ConCORD knows how to execute well. ConCORD dynamically adapts the execution of the query to the amount of redundancy available and other factors. We show that a complex application service (collective checkpointing) can be implemented in only hundreds of lines of code within ConCORD, while performing well.

1. INTRODUCTION

Memory content redundancy, particularly across a large-scale parallel system, represents an opportunity for new services. For example, copy-on-write mechanisms can reduce memory pressure by keeping only a single copy of each distinct page in memory. Fault tolerance mechanisms that seek to maintain a given level of content redundancy can leverage existing redundancy to reduce their memory pressure. Migration of processes or virtual machines (VMs) can leverage identical content at source and destination machines—a single process or VM could be reconstructed using multiple sources. Deduplication of host, VM, process, or application-level snapshots or checkpoints can reduce their storage costs. As far as we are aware, all existing such services integrate the tracking of memory content directly into the service.

We argue that memory content-tracking should be factored out into a separate *platform service* that higher-level *application services*, such as the above, can then be built upon. There are several reasons for this refactoring:

- It will result in having a *single* implementation of memory content-tracking to maintain and enhance.
- There will be no redundant memory content tracking overhead when multiple application services exist.
- The platform service can simplify the creation of application services because their developers will not need to reinvent the wheel.
- The application services benefit as the platform memory content-tracking service is enhanced, for example as new hardware features are leveraged.

To evaluate this concept, we have developed a memory content-tracking platform service called ConCORD. ConCORD is a distributed system that runs across the nodes of a parallel computer, tracks memory content across collections of *entities* (objects that have memory such as hosts, VMs, processes, and applications), and answers queries about this content. In this work we focus our discussion of entities on processes and VMs. The overall ConCORD design and implementation is described in Section 3.

ConCORD has an extensive node-level and collective content-sharing query interface for both memory content and metrics about it, and it is possible to write an application service that uses this interface to make queries. However, we found that many application services could be best understood *as* queries instead of as *users* of queries. For example, a checkpoint of a collection of entities can be thought of as a query for the distinct content they collectively contain. Furthermore, the distributed execution of a collective query in ConCORD already addresses many of the challenges of building an application service, for example, parallelism, scheduling, and synchronization.

The response to these observations is a key concept and contribution of ConCORD and our work, namely the *content-aware service command architecture.* The content-aware service command architecture consists of a carefully designed query template, the content-aware service command, that is parametrized by a set of node-local callback functions that define a specific application service. An analogy can be made

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS-0709168, the Department of Energy (DOE) via grant DE-SC0005343, and Sandia National Laboratories through the Hobbes Project, which is funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the DOE Office of Science. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

HPDC'14, June 23–27, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-2749-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2600212.2600214.

with relational databases: the application service is implemented as a set of stored procedures that are used in a single query, the service command. Although multiple service commands are possible, we have carefully designed the initial service command to be general enough to support a range of application services, including those described above.

Given the very nature of the consensus problem in distributed systems, ConCORD is a best-effort platform service. It is possible that the distributed database's view of the memory content is outdated when a query executes: particular entities may no longer hold certain content hashes, and may hold others that ConCORD is unaware of. This is not an issue for collective query execution, as the service is defined to return best-effort results, but application services are usually not best-effort. To address this, the content-aware service command architecture has a two phase execution model in which the best-effort distributed information is combined with reliable local information. In the first phase, the best effort information is used and inaccuracies are discovered. In the second phase, these inaccuracies are then fed back to be handled using node-local information only. If there is considerable memory content redundancy and few inaccuracies, ConCORD's execution of the content-aware service command implicitly leverages the redundancy without any explicit action on the part of the application service developer. If memory content redundancy is low or there are many inaccuracies due to rapidly changing memory content, ConCORD's execution of the content-aware service command nonetheless remains correct. A single distributed component handles both collective queries and content-aware service commands. Details of the content-aware service command architecture are given in Section 4.

We evaluate ConCORD and the content-aware service command architecture in several ways. First, the small scale and complexity of the ConCORD implementation has bearing on our claim. The bulk of ConCORD consists of ∼12,000 lines of user-level C, with the interface code for a given kind of entity being about 3,000-4,000 lines of C. This suggests it is quite feasible to factor out memory content-tracking as a platform service—the resulting service is not inordinately large. The second aspect of our evaluation is the performance of ConCORD, which we do at scales of up to 128 nodes, considering its overhead, update rates, latencies, and the response time of queries and the service command. Section 5 presents these results.

To test the service command architecture, we describe the design, implementation, and evaluation of an application service. *Collective checkpointing*, described in detail in Section 6, saves the memory content of a collection of entities (we specifically study processes) with the goal of saving each distinct memory block exactly once. This service is implemented on top of ConCORD as a service command that comprises a mere *230 lines* of C code and performs well.

The contributions of our paper are as follows.

- We show it is feasible and sensible to factor memory content-tracking into a separate platform service.
- We define the content-aware service command architecture, which can greatly facilitate the creation of application services through the insight that services are often well expressed as queries, and, indeed that perhaps a single, parametrized query, *the* content-aware service command, is sufficient for many of them.
- We design, implement, and evaluate ConCORD, a proof-of-concept memory content-tracking platform service.
- We build and evaluate a complex application service, collective checkpointing, within ConCORD, providing evidence of the effectiveness of the above.

2. RELATED WORK

Content-based memory sharing has been studied for many years with the goal of deduplicating memory pages to reduce memory footprints or communication costs. For example, VMware ESX [19] and Xen [10] use background hashing and page comparison to transparently identify identical pages in VMs on the same node. Potemkin [18] uses flash cloning and delta virtualization to support a large number of mostly-identical VMs on the same node. Satori [14] implements memory sharing in Xen by detecting deduplication opportunities when reading from a block device. Difference Engine [8] demonstrates even higher degrees of content sharing can be obtained by sharing portions of similar pages. Kernel SamePage Merging (KSM) [3] allows the Linux kernel to share identical memory pages amongst different processes. In the context of HPC systems, SBLLmalloc [5] can identify identical memory blocks on the same node and merge them to reduce the memory usage. In the context of cloud computing, Memory Buddies [20] uses memory fingerprint to discover VMs with high sharing potential and then co-locates them on the same node. Live gang migration [7] optimizes the live migration of a group of co-located VMs on the same node by deduplicating the identical memory pages across them before sending them. VMFlock [2] and Shrinker [16] present similar migration services optimized for cross-datacenter transfer.

Our work differs in two ways. First, we factor memory content tracking into a separate service on top of which application services such as these could be built. We believe we are the first to propose and evaluate this idea, producing a proof of concept, ConCORD. Second, we focus on the whole parallel machine, considering *inter-node* as well as intra-node sharing of content. Our earlier paper [23] and the work of others [5, 12] have clearly shown that such content sharing exists in the HPC context. The work we present here illustrates how to leverage it.

A core mechanism in ConCORD is a custom, high-performance, lightweight, zero-hop distributed hash table (DHT) that associates content hashes with sets of entities and their nodes. Many DHT implementations have been developed [17, 6], with most targeted at wide-area distributed systems. More recent work has considered DHTs for more tightly coupled HPC environments, examples including C-MPI [21] and ZHT [13]. ConCORD's DHT is similarly designed for zero-hop routing and low churn. However, it is not a general key-value store, but rather is specialized specifically for the best-effort content hash to entity set mapping problem in ConCORD. A detailed description is available elsewhere [22].

We use checkpointing as an example application service to test ConCORD and our claims of the utility of the content-aware service command architecture. Checkpointing has a large literature. Nicolae and Cappello describe this literature in detail in their paper on their memory access pattern-adaptive incremental checkpointing system, AI-Ckpt [15]. Our collective checkpointing service saves the content of a group of entities (processes, VMs), deduplicating pages that are have shared content. Other than building upon

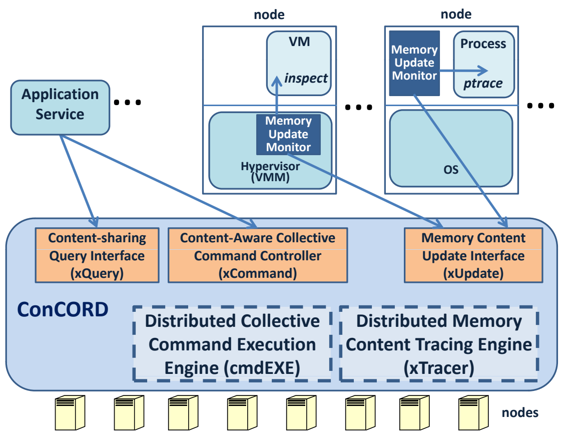

node
VM
inspect
Memory Update Monitor
Hypervisor (VMM)
node
Memory Update Monitor
Process
ptrace
OS
Application Service
Content-sharing Query Interface (xQuery)
Content-Aware Collective Command Controller (xCommand)
Memory Content Update Interface (xUpdate)
ConCORD
Distributed Collective Command Execution Engine (cmdEXE)
Distributed Memory Content Tracing Engine (xTracer)
nodes

Figure 1: ConCORD's high-level architecture. Dashed boxes represent distributed components.

a general-purpose memory content tracking system and the content-aware service command architecture, we do not claim innovations in checkpointing.

3. DESIGN AND IMPLEMENTATION

The goal of ConCORD's design was to factor scalable memory content tracking across the nodes of a parallel computer into a distinct platform service that would be useful for implementing application services and ease such implementation. It was clear from the outset that this required that ConCORD itself be a distributed system.

Initially, we focused on tracking the content of distributed VMs running in instances of our Palacios VMM [11], but almost all of the concepts involved in building VMM-based application services readily generalize beyond VMs. Consequently, it made the most sense to partition ConCORD into a component that operates over general entities (as described in the introduction). The design evolved into one that is partitioned into an entity-agnostic user-level component (the majority of the system), and node-specific components that handle specific kinds of entities on specific kinds of nodes. The latter comprises little code, and may or may not be user-level.

Our original model was that ConCORD would be a distributed database system that captured a best-effort view of memory content across all the nodes and entities. It would then facilitate the creation of application services by providing a query interface that supported both queries that could be answered using information on a single node, and queries that required information to be integrated across multiple nodes, perhaps all of them. These latter collective queries are the key challenge, both for ConCORD to implement and for an application service to use. The content-aware service command architecture is essentially a carefully designed collective query into which the application service is integrated. Instead of the application service making queries, there is a single query that makes callbacks to the application service.

3.1 High-level architecture

Figure 1 illustrates the high-level architecture of ConCORD, highlighting core components and interfaces.

The core functionality of ConCORD is to track memory content across entities in the site. To do so, ConCORD needs to know when the content of a block of memory[1] changes. Furthermore, the nature of the content of a memory block on any node must be accessible (within a query) from all other nodes. To make this possible, ConCORD has two components: (1) a *memory update monitor* running on each node, and (2) a site-wide *distributed memory content tracing engine* that maintains a scalable, distributed database that, given a content hash, can find nodes and entities that are likely to have copies of the corresponding content.

Several kinds of memory update monitors are possible. The figure illustrates two, a kernel-level/VMM-based monitor that inspects a VM's guest physical memory, and a user-level monitor that inspects a process's address space using ptrace. Memory update monitors produce the heartbeat of ConCORD: discovery of memory content changes. One mode of operation for a memory update monitor is to periodically step through the full memory of the entities being traced, identifying memory blocks that have been updated recently, and then sending memory content hashes for the newly updated blocks to the ConCORD memory tracing engine. We use this mode of operation in this paper. In addition, a memory update monitor maintains a local mapping table that allows ConCORD to efficiently locate a memory block's content from its hash.

Memory update monitors can also operate in a mode where they detect writes by leveraging the paging infrastructure. For example, for Palacios VMs, we can apply a copy-on-write model, temporarily marking shadow or nested page table entries as unwritable. Page faults then indicate writes. We can also exploit the x86's nested page table entry's dirty bit, periodically marking page table entries as clean and then rescanning the entries to see which ones the processor has marked dirty. More details about both of these techniques are given elsewhere [4].

Regardless of its kind or mode, a memory update monitor can also be throttled, limiting the rate at which it produces updates. This makes it possible to limit the load placed on the individual node and on the network, trading off load and precision/accuracy, as we described in earlier work [23].

The distributed memory content tracing engine is a site-wide distributed system that enables ConCORD to locate entities having a copy of a given memory block using its content hash. It also allows ConCORD to find the amount of memory content sharing among nodes . ConCORD employs a custom, high-performance, lightweight, zero-hop distributed hash table (DHT) to store unique memory content hashes and map each content hash to a list of entities and their nodes that have a copy of the corresponding memory block. A detailed description of the design of the engine is given elsewhere [22].

The *memory content update interface* is the interface between memory update monitors and the distributed content tracing engine. It is carefully defined so that new monitors can be created, and so that the engine is not entity-specific.

Application services or other tools can ask questions or issue content-aware service commands that operate over the information stored in the tracing engine. The *content-sharing query interface* makes possible two forms of questions. "Node-wise" queries involve data stored within a single node. Note that because memory block content information is stored in

[1]Block size is a configurable parameter, but, as we showed earlier [23], the base page size (4 KB on x64) works very well. We use 4 KB pages throughout the paper.

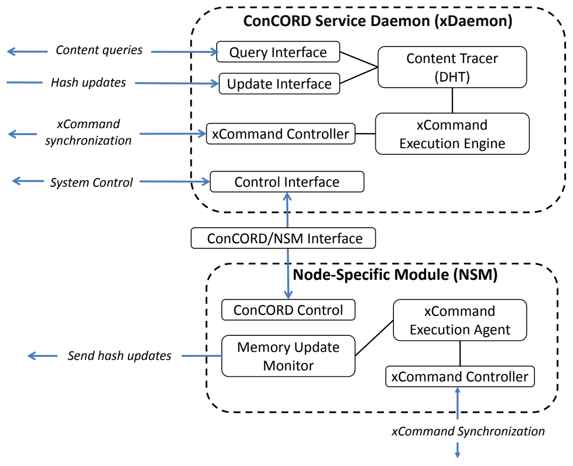


Figure 2: ConCORD's node-level architecture.

a DHT in a node determined by its content hash, the information available in a single node represents a "slice of life" of all the nodes in the system. The query interface also exposes collective queries, which aggregate memory content information across multiple nodes (or all nodes). We describe the query interface (and update interface) in Section 3.3.

The *content-aware collective command controller* provides the interface for both collective queries and content-aware service commands and controls the overall execution of them. The *distributed collective command execution engine* is the most complex distributed component of the system, and is responsible for both these queries and commands. At a high-level, it can be viewed as a purpose-specific map-reduce engine that operates over the data in the tracing engine. We describe these in detail in Section 4.

3.2 Node-level architecture

Figure 2 shows ConCORD's node-level design, which comprises two components and an interface between them.

The *ConCORD service daemon* is a multithreaded user-level component that most importantly includes the local instances of the content-tracing and command execution engines. The daemon exposes multiple interfaces on the network for issuing and responding to queries and updates, synchronizing the execution of collective queries and service commands, and overall system control.

The *node-specific module* (NSM) is responsible for handling particular kinds (e.g., VMs, processes, etc) of entities on the node. We have already seen the most important element of the NSM, namely the memory update monitor, but it also needs to contain some components of the content-aware service command architecture. One aspect of this is how memory is to be accessed for the given kind of entity. Another is the environment in which the callbacks made by the service command need to execute. For example, if we build a checkpointing service for processes, the callback code may run at user-level and use system calls, but if we build the service for VMs, the callback code may run at kernel-level and use VMM-specific mechanisms.

The NSM is also responsible for maintaining a mapping from content hash to the addresses and sizes of memory blocks in the entities it tracks locally. This information is available as a side effect of the memory update monitor.

Node-wise Queries	
number	num_copies(content_hash)
entity_set	entities(content_hash)
Collective Queries	
number	sharing(entity_set)
number	intra_sharing(entity_set)
number	inter_sharing(entity_set)
number	num_shared_content(entity_set, k)
hash_set	shared_content(entity_set, k)
Updates	
void	insert(content_hash, entity)
void	remove(content_hash, entity)

Figure 3: Query and update interfaces.

3.3 Queries and updates

Queries to ConCORD take two forms. *Node-wise queries* depend only on information stored in a single ConCORD instance on a single node, while *collective queries* depend on information spread over instances on multiple nodes, or even all nodes. Application services can issue queries, and NSMs can issue updates through a set of simple C interfaces provided by libraries. Figure 3 shows the query and update interface of ConCORD.

The node-wise queries allow the application service to ask how many copies of the content corresponding to a content hash exist and which set of entities currently have copies of the content. Note that an entity may have more than one copy of given content. An application service can use these queries to find the existing redundancy of particular memory content, and where the replicas are.

The first three collective queries compute the degree of content sharing (redundancy) that exists across a set of entities. Here we are concerned about all the content in these entities, not particular content. We consider two forms of sharing. Intranode sharing is sharing of content across the entities located within a node, while internode sharing is sharing of content across entities in distinct nodes. Either form, or both together can be queried. An application service can use these queries to discover if there is sufficient overall redundancy to make it worthwhile to leverage.

The final two collective queries allow an application service to discover, for a set of entities, the amount of content that is replicated k or more times, and the set of content hashes for which this is true. These "at least k copies" queries allow the application service to find content whose redundancy is particularly useful to leverage due to its many replicas.

Updates simply insert or delete (key, value) pairs, where the key is a content hash and the value is the entity. A hash over the key determines the node and service daemon to which the update is routed. The target daemon maintains a hash table that maps from each content hash it holds to a bitmap representation of the set of entities that currently have the corresponding content. Given this simple mapping, the originator of an update can not only readily determine which node and daemon is the target of the update, but, in principle, also the specific address and bit that will be changed in that node.

3.4 Communication

ConCORD uses two forms of communication. First, reliable 1-to-n broadcast and synchronizing communication is used between client libraries and the shell, and xDaemon and VMM instances. This communication is infrequent, generally only occurring when a query or service command is running. The data size for messages here is small. Unreliable peer-to-peer data communication is used among service daemons and NSMs. Examples include an update being sent from an NSM to a target service daemon, and a content hash exchange among service daemons during the execution of a content-aware service command. The second form of communication, unreliable peer-to-peer communication, forms the bulk of communication in the system, both due to the frequency of such communication and its volume. The motivation for separating these communication paths and their requirements is to eventually facilitate the use of fast, low-level communication primitives for most communication. For example, because the originator of an update in principle knows the target node and address, and because the update is best effort, the originator could send the update via a non-blocking, asynchronous, unreliable RDMA.

In its current implementation, ConCORD uses UDP for network communications in both cases. Unreliable peer-to-peer communication is done using standard UDP socket calls, i.e., "send and forget". Reliable 1-to-n communication is implemented on top of standard UDP, combined with an acknowledgment protocol that allows out-of-order delivery of ConCORD messages. We require the underlying hardware to provide error detection for both reliable and unreliable communication. As we are operating within a single network of a parallel machine, underlying hardware mechanisms like the Ethernet CRC readily do this. Reliable messages may arrive out of order, but they *will* arrive without errors. Unreliable messages may not arrive, but if they do arrive, they have no errors.

4. CONTENT-AWARE SERVICE COMMAND

The goal of the content-aware service command architecture is to ease the construction of application services that build on ConCORD's distributed memory content tracing facility. As we have described so far (Section 3), ConCORD helps the application service developer avoid reinventing one wheel, namely that of memory content tracking and queries. In the design of the content-aware service command architecture, we seek to help the developer avoid reinventing another wheel, the distributed execution of the application service over the relevant memory content.

The developer creates the service by implementing a set of callbacks. The interface of these callbacks and the protocol by which they are invoked form the core of the service command architecture from the developer's perspective. The callbacks parametrize a service command. The parametrized service command is the application service implementation. The execution system in ConCORD in turn is able to automatically parallelize and execute a service command over the nodes of the parallel machine. The execution is driven by the memory content tracked in ConCORD, as well as the "ground truth" memory content information available locally. As a consequence, the service can leverage redundancy while being correct.

4.1 Running example

Consider that we seek to build a "collective checkpointing" service that operates over a set of distributed processes (or VMs), where each process's address space consists of pages. When we execute the service, we require correctness: each page in each process must be recorded so that it can be restored later. We also desire efficiency: The checkpoint should contain few duplicate pages; ideally, each distinct page of content would be recorded exactly once.

A correct implementation would simply record each page in each process. A highly efficient implementation would find each distinct page captured in ConCORD's database, select a replica, and have that replica recorded. However, ConCORD's database is best effort, as described previously, so this implementation would not be correct—it would include pages that no longer exist, and fail to include pages that aren't yet in the database. What the service command architecture allows us to do is combine efficiency (using ConCORD's database) and correctness (using local information). We will describe the design, implementation, and evaluation of this service in detail in Section 6.

4.2 Scope and mode

A service command has a scope, a set of entities over which it will operate. For example, this might be the set of processes or VMs we wish to checkpoint. We refer to these as the set of *service entities* (SEs), or we say that these entities have the service role. Because ConCORD's memory tracking is not specific to any one application service, however, it is likely that there are many other entities being tracked that could contribute to our service. For example, an independent process on a separate node might share a page of content with one of the processes we are checkpointing. If the separate node wrote this page, it would speed up our checkpoint. We refer to entities that will be involved in this way as *participating entities* (PEs), or we say that these entities have the participant role. The scope of the execution of an application service consists of the set of SEs and PEs. The service command uses the memory content in the SEs and PEs to apply the service to the SEs.

A service command can be executed in one of two modes. In *interactive mode*, the application service's callbacks are invoked for each content hash and are expected to immediately apply the relevant transformations. In *batch mode*, the callbacks instead drive the creation of an execution plan by the application service. The application service then executes its plan as a whole. This allows the application service developer to refine and enhance the plan. In this paper we focus on interactive mode.

4.3 Interface and execution

Figure 4 describes the high-level interface of an application service. The detailed C interface is available elsewhere [22, Chapter 7]. The service developer implements the application service by implementing these callbacks. The service command invokes the callbacks in four phases, as shown in the figure. At the heart of execution are the collective phase (which uses the content information in ConCORD's database) and the local phase (which uses node-local content information).

Service initialization passes a service-specific configuration file to be parsed. Within the service_init() function, which is executed on each node holding a service or participating

Service Initialization	
error	service_init(service, config)
Collective Phase	
error	collective_start(entity_role, entity, content_hash_set, private_service_state)
entity	optional_collective_select(content_hash, entity_set, private_service_state)
error	collective_command(entity, content_hash, pointer, size, private_service_state)
error	collective_finalize(entity_role, entity, content_hash_set, private_service_state)
Local Phase	
error	local_start(entity_role, entity, content_hash_set, private_service_state)
error	local_command(entity, content_hash, pointer, size, private_service_state)
error	local_finalize(entity_role, entity, content_hash_set, private_service_state)
Service Teardown	
error	service_deinit(private_service_state)

Figure 4: Content-aware service command callbacks. An application service is created by implementing these callbacks.

entity, the developer can also associate private service state with the service. This is then passed to subsequent callbacks on that node as the private_service_state parameter. This state can grow during the execution of the command.

The collective phase of execution is supported by four callbacks, one of which is optional. The collective_start() function is executed exactly once for each service and participating entity. It indicates what role (service or participating) the entity has and provides a partial set of the content hashes ConCORD believes the entity contains. This partial set is derived using the data available on the local instance of the DHT, and is mostly advisory. collective_start() is usually where the service typically allocates and initializes resources that are outside of the scope of ConCORD. For example, the collective checkpointing service opens its checkpoint files here.

ConCORD synchronizes on collective_start(). After all instances have returned, it will then compute distinct content hashes across all of the service entities, and, for each of them, find the set of service entities and participating entities that appear to have this content. This information will drive callbacks to the PEs and SEs to execute over their content. Because a particular content hash might exist on multiple entities, ConCORD needs to pick one. If the service developer has implemented the optional_collective_select() callback, ConCORD invokes this on some node to allow the service to choose. Alternatively, it chooses one of the entities at random.

Given a content hash and a selected entity, ConCORD invokes the collective_command() callback on the node where the entity resides. In addition to the hash and entity, ConCORD also passes an opaque pointer and data size to the callback. This information is computed by the NSM (Section 3.2). In effect, in the collective phase of execution, ConCORD maps the collective_command() function across unique content hashes that it believes exist in the SEs, selecting for each hash a single entity (and thus node) from the SEs and PEs that it believes contains that hash. A collective_command() invocation may fail because the content is no longer available in the node. When this is detected, or if the collective_command() invocation is taking too long, ConCORD will select a different potential replica and try again. If it is unsuccessful for all replicas, it knows that its information about the content hash is stale.

After exhausting all relevant content hashes in its distributed database, ConCORD invokes the collective_finalize() function for each entity. Here, the service has a chance to reduce and gather results from work that has been done during the collective phase, and clean up and free resources that are not needed in the local phase. collective_finalize() also acts as a barrier.

ConCORD now invokes local_start() for each SE to allow the service to prepare for the local phase. PEs are not involved in the local phase. ConCORD then invokes the local_command() callback for each memory block (e.g., page) in each service entity. The callback includes the block's hash, as well as the set of all hashes that have been successfully handled by a previous collective_command(). The service can thus easily detect and handle content that ConCORD was unaware of. For example, the collective checkpoint service can save a page that was not previously saved. Further, for content that ConCORD did previously handle, the service can now make use of that fact. For example, the collective checkpoint service can save a file offset for a page that was previously saved. With successive invocations for the process, the collective checkpoint service builds a file of pointers to where the process's page content exists.

The local_finalize() callback is invoked for each service entity to complete the local phase and to synchronize. Here, the service typically does cleanup, for example, closing the checkpoint files.

At the end of service command execution, the service_deinit() function is called on each node with a service or participating entity. This function is responsible for interpreting the final private service state to indicate to ConCORD whether the service was successful or not. For batch mode execution, the service typically builds up a plan in the private service state, driven by collective_command() and local_command() callbacks, and then executes it as part of local_finalize() or service_deinit().

5. GENERAL EVALUATION

We now describe the general evaluation of ConCORD, independent of any application service. The evaluation focuses on ConCORD's overheads and scalability. ConCORD consists of ∼19,300 lines of code (∼12,000 in the core system, ∼3,300 in NSM for processes, ∼4,000 in NSM for VMs).

5.1 Testbeds

Our evaluations are done on the following hardware.

Old-cluster is used for most of the general evaluation of this section. It consists of 24 IBM x335 nodes, each having two dual-core Intel Xeon 2.00GHz processors, 1.5 GB RAM, 32 GB disk, and a gigabit NIC connected to a 100 Mbit Cisco 3550 48 port switch, which has the full backplane bandwidth needed to support all of its ports. The nodes run Red Hat

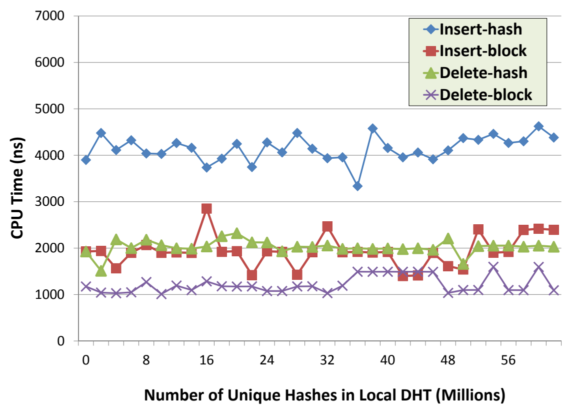


Figure 5: CPU time of DHT updates as a function of number of unique hashes in the local node.

Enterprise 6.2 with a 2.6.30.4 kernel. Note that good performance on such old hardware bodes well for newer hardware.

New-cluster is used for benchmarking the DHT. It consists of 8 Dell R415 machines, where each has two quad-core 2.2 GHz AMD Opteron 4122 processors, 16 GB RAM, 500 GB disk, and a gigabit NIC attached to an HP Procurve 2848 48 port gigabit switch, which has the full backplane bandwidth needed to support all of its ports. The nodes run Fedora 15 with a 2.6.38 kernel.

Big-cluster is used to study high node scaling of the service command and the collective checkpoint service (Section 6). *Big-cluster* is Northwestern's HPC resource, consisting of 824 nodes in three categories. 504 nodes contain two quad-core 2.4 GHz Intel Nehalem processors and 48 GB of RAM, and are connected via an InfiniBand DDR network. Two other machine pools contain nodes with faster processors and similar RAM that are interconnected with QDR InfiniBand (252 nodes) and FDR10 InfiniBand (68 nodes). They run Red Hat 6.4 with a 2.6.32 kernel.

5.2 Distributed memory content tracing

We expect the memory tracing component of ConCORD, which discovers memory content changes and maintains the distributed database of content to have low overhead. Here, we focus in particular on the overheads of the distributed database (the DHT). We do not consider the amount of redundancy that is available within particular environments and how well this can be captured because our previous publication [23] and Xia's dissertation [22] include this. The latter presents a detailed study of redundancy in a range of parallel applications.

Our previous publication also considered the costs and overheads of the memory update monitor, which scans an entity's memory to discover updates. We summarize these results. On our slowest hardware (*Old-cluster*), we found that a memory update monitor that periodically scans a typical process from a range of HPC benchmarks and computes content hashes from its pages exhibits a 6.4% CPU overhead when scanning every 2 seconds, and a 2.6% CPU overhead when scanning every 5 seconds. This is with the MD5 hash function. With the non-cryptographic SuperHash function, these overheads drop to 2.2% and less than 1% for these rates. The updates sent into the network from the memory update monitor typically require about 1% of the outgoing link bandwidth.

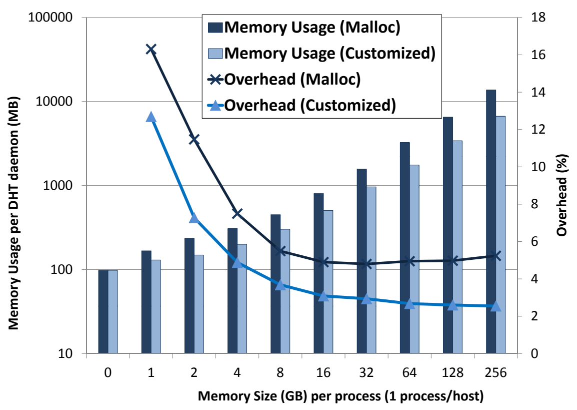


Figure 6: Per-node memory use to store the DHT as a function of the size of entities

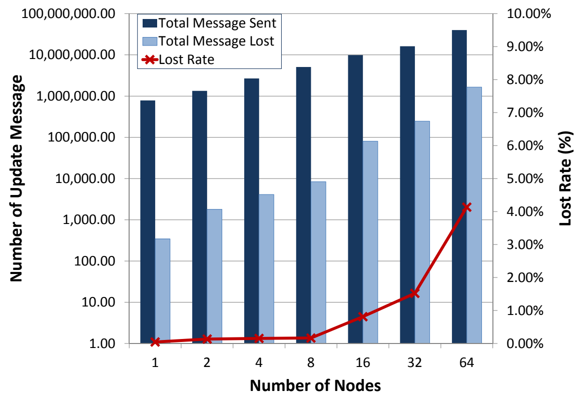


Figure 7: Update message volume and loss rate as a function of the number of nodes on *Big-cluster*.

We now consider the costs of updates to the distributed database, which are measured on *New-cluster*.

One question is the latency of updates, which depends on both the network and local computation. At the network level, the latency of an update is the latency of a UDP packet. In Figure 5, we illustrate the costs of the local computation as a function of the number of hashes stored within the local instance of the DHT. Both inserts and deletes are measured. The block insertion or delete is the cost of updating the local mapping from content hash to block within the entity, while the hash insertions and deletions are the cost of updating the (typically remote) mapping from content hash to entity. The important thing to note is that these costs are independent of how many unique content hashes are stored.

A second question is the memory costs of storing the content of the DHT. Figure 6 shows the memory needed on each node as the entity size grows. Because the allocation units of the DHT are statically known, a custom allocator can improve memory efficiency over the use of GNU malloc. With the custom allocator, at an entity memory size the same as the node's physical memory (16 GB), the additional memory needed is about 8%. Even at 256 GB/entity (achieved using swapping), only about 12.5% more memory is needed to store the DHT content.

Finally, we consider the network load imposed by updates, and how often updates are lost. Here, we use *Big-cluster* to consider a larger number of nodes. Figure 7 shows the results. Here each node contains one entities (4 GB total

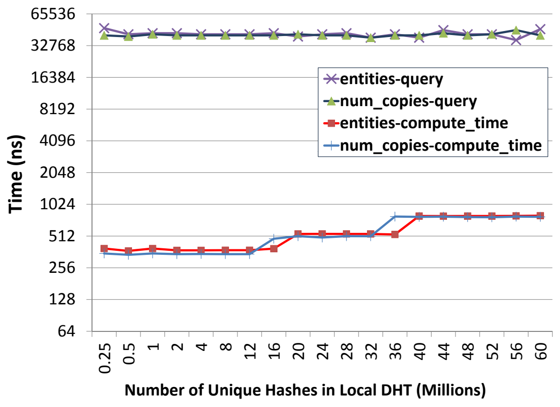


Figure 8: Latency of node-wise queries as a function of unique hashes in the local node.

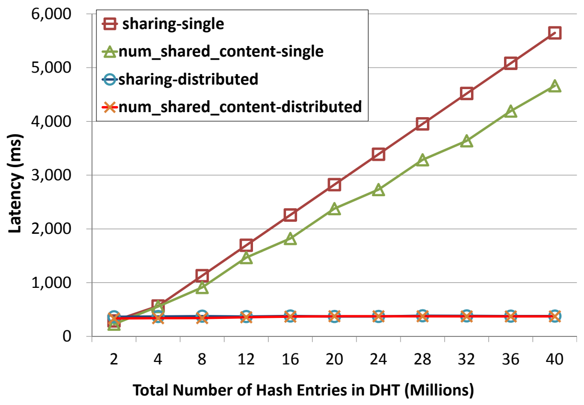


Figure 9: Total latency for collective queries as a function of the number of content hashes.

RAM per entity), and we are considering the initial scan of their memory, that is, each node is sending an update for each page of each entity, which is the worst case. As we scale up the number of nodes the total number of update messages in this scenario of course scales linearly with it. However, so do the number of sources and destinations of these messages, as the DHT itself grows. We are currently trying to understand why the loss rate grows with scale.

Our results suggest that the memory update monitor and updates to the distributed database scale well with the number of nodes and the total size of entities across those nodes, in terms of the update work and network traffic per node.

5.3 Queries

We now consider the performance of queries in ConCORD. Here, *Old-cluster* is used. Our primary focus is on latency as the system expands in terms of amount of memory tracked and number of nodes participating.

We first consider the node-wise queries, those that require only a single node to respond. Figure 8 illustrates the latency for the two node-wise queries. The "query" time includes the communication and computation, while "compute_time" captures the time for the computation at the answering node. The latency is dominated by the communication, which is essentially a ping time. This should come as no surprise as these queries are computed with a local hash table lookup to a bitmap that is then scanned.

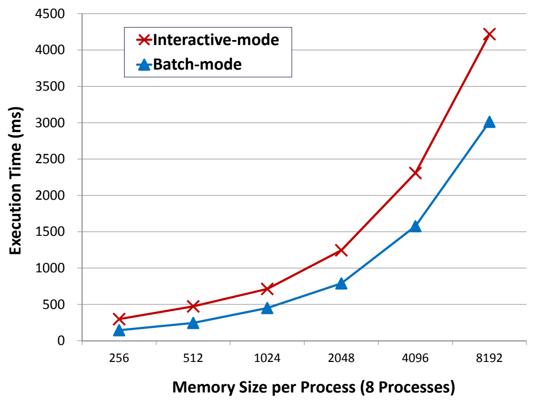


Figure 10: Null service command execution time on a fixed number of SEs and nodes with increasing memory size per process.

When observed from the perspective of a single node, the collective queries perform similarly to the node-wise queries. More important is the latency seen across the machine as communication is used to compose the query results. Figure 9 shows the total latency for the collective queries. For the "distributed" case, the DHT (and thus the query computation and communication) is distributed over the nodes of *Old-cluster*. Here, we scale up the number of nodes used so that the number of content hashes per node is kept constant (at about 2 million hashes per node). For the "single" case, the DHT (and thus the computation) is limited to a single node, which handles more and more hashes. As can be seen from the figure, at around 2-4 million hashes there is a crossover point, and the distributed storage and execution model perform better, giving a constant response time as the system scales.

Under the expected operating model of a system like ConCORD, the maximum possible number of content hashes is limited by the total memory of the parallel machine. As we add more nodes, we add more memory that may need to be tracked and operated over, but we also add more compute and network capability to support tracking and operating over it. The figure makes it clear that the system scales in terms of memory and nodes, providing a stable response time as this happens. On *Old-cluster*, our slowest computational and network hardware, this stable response time for collective queries is about 300 ms.

5.4 Null service command

We now consider the performance of the content-aware service command. Here, we focus on the baseline costs involved for any service command by constructing a "null" service that operates over the data in a set of entities, but does not transform the data in any way. That is, all of the callbacks described in Figure 4 are made, but they do nothing other than touch the memory. We consider this command both in interactive mode and in batch mode. In batch mode, the callbacks record the plan and in the final step the memory is touched. Evaluations of both modes are done on *New-cluster*, while on *Big-cluster*, we study the scaling of interactive mode.

Figure 10 shows the execution times for the null service command for a fixed number of SEs (processes) and nodes as we vary the memory size per SE. As we would hope,

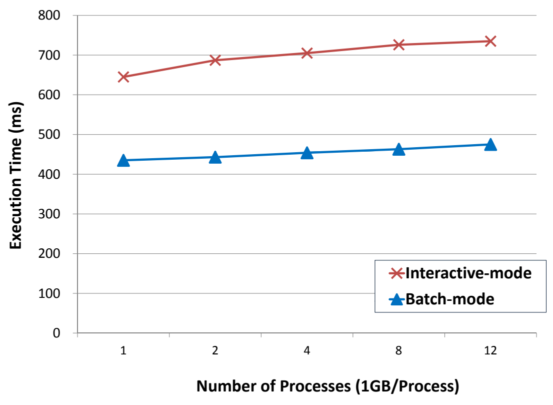


Figure 11: Null service command execution time for an increasing number of SEs and nodes, holding per-SE memory constant.

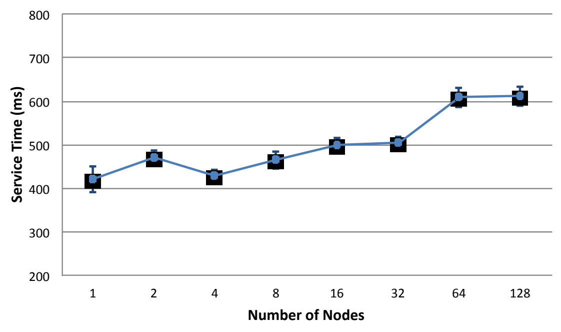


Figure 12: Null service command response time on *Big-cluster*.

the execution time is linear with the total memory size of the SEs. Figure 11 shows the execution time for the null service command as we simultaneously increase the number of SEs (processes) and nodes, while holding the per-process memory constant. This is how we would expect to use the service command architecture in practice—as we scale the number of SEs over which the service command operates, we expect the number of nodes to also scale, as there is a limited amount of memory in each node. In this regime, we see that the execution time stays relatively constant, suggesting the service command architecture scales nicely.

Beyond this scalable execution time, we would also hope to see that the network traffic to execute the null service command scales. As we scale the number of SEs and nodes simultaneously, as we did for Figure 11, the average traffic volume sourced and sinked per node stays constant at about 15 MB. This further lends support to the scalability of the service command architecture under the expected regime.

Figure 12 shows the scaling of the null service command running in interactive mode on *Big-cluster*. As previously, we scale up nodes and total memory simultaneously. The response time is constant, up to 128 nodes, providing further evidence of the scalability of the content-aware service command architecture.

6. COLLECTIVE CHECKPOINTING

We claim that the content-aware service command is sufficient for implementing a range of application services. To evaluate this claim, we considered three services. The first, collective checkpointing, is described in detail here. The second, collective VM reconstruction, recreates the memory image of a stored VM (the service entity) using the memory content of other VMs currently active (the participating entities). This is described in the first author's dissertation [22, Section 7.2]. The third, collective VM migration, migrates a group of VMs from one set of nodes to another set of nodes, leveraging memory redundancy.

Our focus in collective checkpointing (and for the other two) is in exposing and leveraging memory redundancy. Our work does not extend to, for example, selecting which replica to use for a specific operation. ConCORD, through the optional_collective_select() callback, provides the ability for the service developer to introduce such smarts, however.

The goal of collective checkpointing is to checkpoint the memory of a set of SEs (processes, VMs) such that each replicated memory block (e.g., page) is stored exactly once. For parallel workloads that exhibit considerable redundancy (for example, the Moldy and HPCCG benchmarks we studied in previous work [23]), collective checkpointing will reduce the overall size of the checkpoint and the benefits will grow as the application scales. Such benefits should be clear whether we checkpoint processes or VMs containing them.

The collective checkpointing service is easily built within the content-aware service command architecture. The implementation we use here comprises a total of only 230 lines of user-level C and can be seen in its entirety elsewhere [22, Appendix A]. The developer does not have to consider parallel execution, memory content location, or other factors. The performance of the checkpointing service is also promising. When no redundancy exists, the execution time has only a small overhead over the obvious design of purely local checkpoints for each SE. Furthermore, checkpoint size can be reduced significantly when redundancy exists—considerably more than compression. Finally, the time to checkpoint a set of SEs scales well as the number of entities and nodes increases. These results support our claim.

6.1 Design and implementation

Figure 13 schematically shows the format of the collective checkpoint. Only two SEs are shown for simplicity, although a real checkpoint will have as many as there are processes or VMs. A single shared content file stores the content of most memory blocks that have been found. That is, for each distinct memory block with one or more copies among the SEs, there is ideally exactly one copy stored in the shared content file.

Each SE has a separate checkpoint file that contains an entry for each memory block within the SE. The entry either represents the content of that block or it represents a pointer to the content's location within the shared content file. The reason why content may exist within the SE's checkpoint file is due to the best-effort nature of ConCORD. If an SE contains a memory block whose content is unknown to ConCORD, it appears in the SE's checkpoint file, otherwise it appears in the shared content file.

The syntax $1 : E : 3$ means that memory block 1 (e.g., page 1) of the SE holds content whose hash is E and which is stored as block 3 of the shared content file. Figure 13 shows four replicated blocks. Each SE has four blocks, three of which are stored as pointers, and one of which is stored as data. In this diagram, a total of 8 blocks (each SE has 4 blocks) is stored using 6 blocks. Ignoring the pointers, this

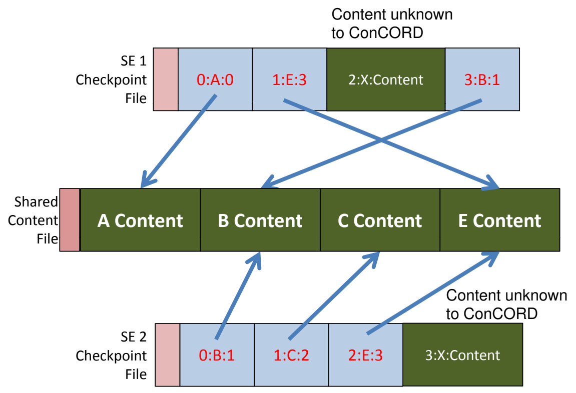


Figure 13: Checkpoint format, showing two SEs.

would yield a *compression ratio* of 75% (6/8) compared to simply saving each SE's blocks independently.

To restore an SE's memory from the checkpoint, we need only walk the SE's checkpoint file, referencing pointers to the shared content file as needed. Our implementation focuses on taking the actual checkpoint. In describing this process, we use Section 4's terminology, in particular the collective and local phases of service command execution.

In the collective phase, ConCORD determines what it believes to be the distinct memory content hashes that exist across the system. It then issues collective_command() callbacks for them. Our handler receives the content hash, as well as an NSM-supplied pointer to the data of the local copy of the content. It then issues an append call to the file system to write the block to shared content file. The file system returns the offset which is stored in a node-local hash table that maps from content hash to offset. It indicates to ConCORD that it has handled the content hash and includes the offset in the private field of its return value. At the end of the collective phase, the shared content file is complete, and each node has its local hash table from which pointers into the shared content file can be derived.

Since these collective commands are happening in parallel across the nodes, we do require that the file system provide atomic append functionality with multiple writers. In effect, we have a log file with multiple writers. This is a well-known problem for other forms of logging on parallel systems and is either a component of the parallel file system or of support software that builds on top of it (e.g., [9]).

In the local phase of execution, ConCORD issues callbacks that cause each SE's memory to be scanned, with a local_command() callback issued for each block. These callbacks write the records of the SE's own checkpoint file. If the callback indicates that the content hash was successfully handled in the collective phase, then the private data field is interpreted as being the offset within the shared content file. The callback can then append a pointer record to the SE's checkpoint file. If the callback indicates that the content hash was not handled during the callback phase, then the block does not exist in the shared content file because ConCORD's DHT was unaware of it. In this case, the callback writes a record that contains the content to the SE's checkpoint file.

After the collective and local phases are done, the full set of files is complete. The assorted callback functions for initialization of the service and the phases are where the files are opened and closed.

6.2 Evaluation

Our evaluation focuses on the overall performance and scalability of the collective checkpointing service. We consider performance on *Old-cluster* and *Big-cluster*, which are described in Section 5.1.

Reducing checkpoint size is an important goal of collective checkpointing, but the degree to which this is possible depends on the workload. In the following, we consider two workloads consisting of SEs that are MPI processes. Moldy [1] is a general purpose molecular dynamics simulation package we identified in our previous paper [23] as exhibiting considerable redundancy at the page granularity, both within SEs and across SEs. In contrast, Nasty is a synthetic workload with no page-level redundancy, although its memory content is not completely random.

We consider four methods for generating the checkpoint. Raw simply has each SE save its content independently. ConCORD is not used. Raw-gzip further concatenates all the per-SE files and compresses them with gzip. ConCORD uses the scheme of Section 6.1. Finally, ConCORD-gzip further compresses the shared content file. The purpose of introducing gzip is to understand how redundancy elimination through compression compares to ConCORD's approach, and how much it can improve the ConCORD approach.

Figure 14 shows the compression ratio for the four strategies, for both Moldy and Nasty, as a function of the number of processes. For Moldy, the measured degree of sharing (the results of the sharing() query of Figure 3 is also shown. These results are measured on *Old-cluster*.

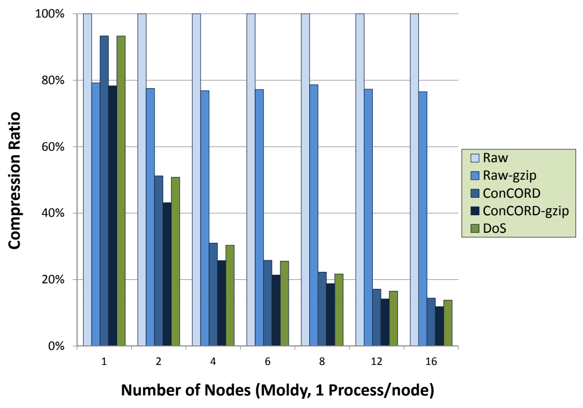
The Moldy results show that when redundancy exists, the collective checkpoint service (ConCORD) is able to exploit it well. Indeed, all the redundancy that ConCORD has detected via its query interface is captured in the service command. The results also show that the redundancy goes well beyond that which can be captured via compression, although compression does slightly increase compression ratio. The Nasty results, in contrast, show that when no page-level redundancy exists, the additional storage overhead of the collective checkpoint service is minuscule.

We measured the response time of the collective checkpoint service and the simple raw service in several ways. In doing so, we factor out the overhead of the file system by writing to a RAM disk. We compare the performance of the raw service with and without gzip with the performance of the interactive ConCORD-based collective checkpoint service as described in Section 6.1. We also consider a batch ConCORD-based collective checkpoint. These results are measured on *Old-cluster*.

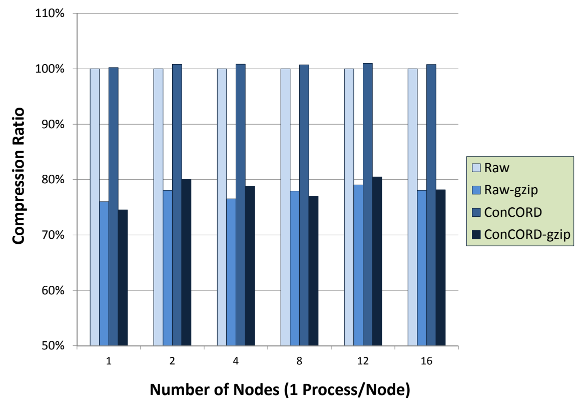
In Figure 15 we fix the number of nodes and SEs while we increase the memory size of each SE. This creates more work per node. The response times of all of the checkpointing implementations increase linearly with the amount of memory we are checkpointing, as we would expect.[2] The collective checkpointing service is faster than raw with gzip, but slower than the simple raw service.

Figure 16 shows results for scaling within the expected regime. Here, the amount of memory per SE is kept constant, and the number of nodes grows with the number of SEs. Essentially, the application uses more nodes to tackle a larger problem. In this regime, the response time is inde-

[2]The plot is on a log-log scale to make it easier to see the differences, but the curves are, in fact, linear.



(a) Moldy (considerable redundancy)



(b) Nasty (no redundancy)

Figure 14: Compression ratios for Moldy and Nasty.

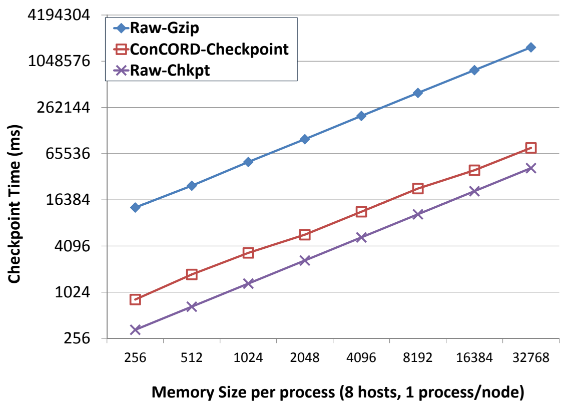


Figure 15: Checkpoint response time for a fixed number of SEs and nodes as the memory size per SE increases.

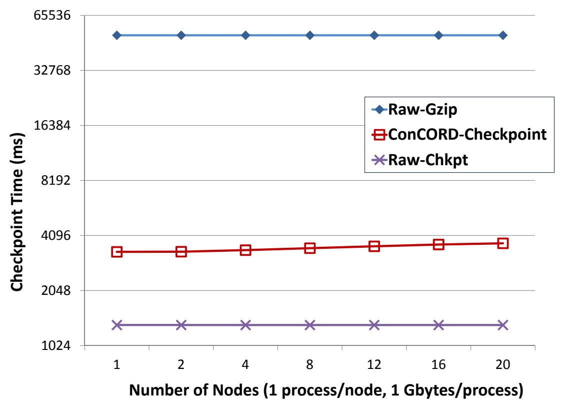


Figure 16: Checkpoint response time as a function of the number of SEs as the number of nodes increases with the memory being checkpointed.

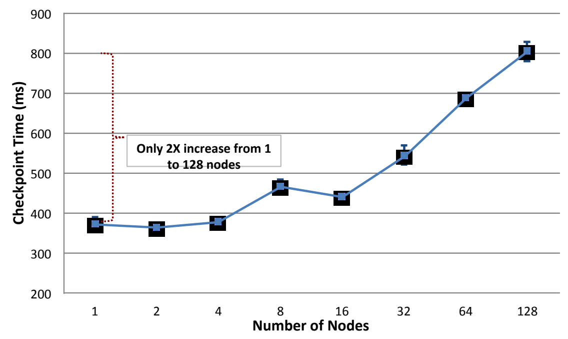


Figure 17: Checkpoint response time on *Big-cluster*.

pendent of the number of nodes. All of the checkpointing approaches scale. Similar to the earlier figure, the response time of the collective checkpointing service is within a constant of the embarrassingly parallel raw checkpointing service. This is evidence that the asymptotic cost to adding awareness and exploitation of memory content redundancy across the whole parallel machine to checkpointing using the content-aware service command is a constant.

Figure 17 shows the scaling of the collective checkpointing service on *Big-cluster*. As previously,we scale up the memory and the number of nodes simultaneously. The response time is virtually constant (within a factor of two) from 1 to 128 nodes, supporting the case that a scalable application service can be effectively built on top of the content-aware service command architecture.

7. CONCLUSIONS

We have made the case for factoring out system-wide memory content-tracking into a separate platform service on top of which application services can be built. ConCORD is a proof-of-concept for such a platform service. In building ConCORD, we also developed the content-aware service command architecture, which allows an application service to be readily built as what is effectively a parameterized query in the system. This greatly simplifies the construction and execution of the application service because it allows us to leverage the platform service's existing automatic and adaptive parallel execution model.

8. REFERENCES

[1] MOLDY. http://www.ccp5.ac.uk/moldy/moldy.html.

[2] Al-Kiswany, S., Subhraveti, D., Sarkar, P., and Ripeanu, M. VMFlock: virtual machine co-migration for the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC'11)* (June 2011).

[3] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium (OLS'09)* (July 2009).

[4] Bae, C., Xia, L., Dinda, P., and Lange, J. Dynamic adaptive virtual core mapping to improve

power,energy, and performance in multi-socket multicores. In *Proceedings of the 21st ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2012)* (June 2012).

[5] Biswas, S., de Supinski, B. R., Schulz, M., Franklin, D., Sherwood, T., and Chong, F. T. Exploiting data similarity to reduce memory footprints. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Systems (IPDPS'11)* (May 2011).

[6] Dabek, F. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2005.

[7] Deshpande, U., Wang, X., and Gopalan, K. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC'11)* (June 2011).

[8] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)* (December 2008).

[9] Ilsche, T., Schuchart, J., Cope, J., Kimpe, D., Jones, T., Knoepfer, A., Iskra, K., Ross, R., Nagel, W., and Poole, S. Enabling event tracing at leadership-class scale through i/o forwarding middleware. In *Proceedings of the 21st ACM International Symposium on High-performance Parallel and Distributed Computing (HPDC'12)* (June 2012).

[10] Kloster, J., Kristensen, J., and Mejlholm, A. On the feasibility of memory sharing: Content-based page sharing in the xen virtual machine monitor. Tech. rep., Master Thesis, Department of Computer Science, Aalborg University, 2006.

[11] Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)* (April 2010).

[12] Levy, S., Ferreira, K. B., Bridges, P. G., Thompson, A. P., and Trott, C. An examination of content similarity within the memory of hpc applications. Tech. Rep. SAND2013-0055, Sandia National Laboratory, 2013.

[13] Li, T., Zhou, X., Brandstatter, K., Zhao, D., Wang, K., Rajendran, A., Zhang, Z., and Raicu, I. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)* (May 2013).

[14] Miłós, G., Murray, D. G., Hand, S., and Fetterman, M. A. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX'09)* (June 2009).

[15] Nicolae, B., and Cappello, F. Ai-ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd ACM International Symposium on High-performance Parallel and Distributed Computing (HPDC'13)* (June 2013).

[16] Riteau, P., Morin, C., and Priol, T. Shrinker: Improving Live Migration of Virtual Clusters over WANs with Distributed Data Deduplication and Content-Based Addressing. In *Proceedings of the 17th International European Conference on Parallel and Distributed Computing (EuroPar'11)* (August 2011).

[17] Stoica, I., Morris, R., Karger, D. R., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM'01* (August 2001).

[18] Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A. C., Voelker, G. M., and Savage, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)* (October 2005).

[19] Waldspurger, C. A. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)* (December 2002).

[20] Wood, T., Tarasuk-Levin, G., Shenoy, P. J., Desnoyers, P., Cecchet, E., and Corner, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 5th International Conference on Virtual Execution Environments (VEE'09)* (March 2009).

[21] Wozniak, J. M., Jacobs, B., Latham, R., Lang, S., Son, S. W., and Ross., R. C-mpi: A dht implementation for grid and hpc environments. Tech. Rep. ANL/MCS-P1746-0410, Argonne National Laboratory, 2010.

[22] Xia, L. *ConCORD: Tracking and Exploiting Cross-Node Memory Content Redundancy in Large-Scale Parallel Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Northwestern University, July 2013. Available as Technical Report NWU-EECS-13-05.

[23] Xia, L., and Dinda, P. A case for tracking and exploiting memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing (VTDC'12)* (June 2012).