Towards Efficient VM Management on Clouds

Bogdan Nicolae

University of Rennes 1, France

July 7, 2010

About me

- PhD Student, 3rd year, KerData Team, IRISA/INRIA Rennes, France
  - Thesis: BlobSeer - A New Vision on Data Management for Large-Scale, Distributed Systems
  - Web: http://blobseer.gforge.inria.fr
- Visiting student: Argonne National Laboratory, USA
  - Working with: John Breshnahan, Kate Keahey

Outline

- Context
- Assumptions
- Approach
- Implementation
- Results
- Conclusions

What is cloud computing?

- Computing as utility rather than capital investment
  - Buy electricity rather than buy generators
- Multiple abstractions: IaaS, PaaS, SaaS
  - IaaS: EC2, Nimbus, Eucalyptus, OpenNebula
  - PaaS: Elastic MapReduce
  - SaaS: Google Apps

Why use cloud computing?

- Advantages
  - Low entry cost
  - Pay only for what you use: CPU time, network traffic, storage space
  - Elasticity
  - Rapid deployment: provider cares for configuration, hardware, etc.
- Disadvantages
  - Security
  - High costs for long term usage
  - Provider lock-in

Distributed applications on clouds

- Typical scenario:
  - Hundreds of nodes work in harmony to solve a problem
  - Each node runs at least one VM
  - VMs are instantiated from a common initial image
- Challenges:
  - Efficient propagation of initial image content
  - Efficient checkpointing/resume
- Efficiency is:
  - High performance
  - Low costs: network traffic, storage space

Cloud infrastructure

- Large number of compute nodes
  - Locally attached storage
  - Limited capacity: hundreds of GB
  - Not persistent
- Relatively smaller number of storage nodes
  - Dedicated storage devices
  - Huge capacity: order of TB
  - Persistent
- Communication model: all interconnected

Application access pattern

- T = total image size, r = amount read, w = amount written

1. Boot the VM from a given image
   - Read kernel, Read config files, write temporary files
   - Pattern: $0 < r << T$ and $0 < w << T$
2. Run user application
   - CPU-intensive or uses external storage: $0 < r << T$ and $0 < w << T$
   - Read-intensive (e.g. input stored in image): $0 << r < T$ and $0 < w << T$
   - Write-intensive (e.g. temp files, log files): $0 < r << T$ and $0 << w < T$
3. Shutdown VM
   - r and w are negligible

Concurrent access to the initial contents

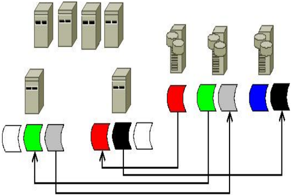
- Boot process
  - Read same parts in same order on each VM
  - Interleaving of CPU-time with I/O time: same parts not read at exactly same time by two different VMs
- Runtime
  - Concurrent access depends on access pattern
- Checkpointing
  - Concurrent write access pattern to storage nodes
  - Depends on size of local modifications

Checkpointing: How to save application state

- Two possible ways of expressing the application state:
  1. Explicitly
     - Application saves state as temporary files in the image
     - Save only what is needed
     - Application resumes from temporary files
  2. Implicitly
     - Save RAM, CPU registers, sockets, etc. for all images
     - Potentially large amount of storage space
     - Global state is not the sum of states of all devices of VMs
- For this work: case 1 (state is sum of local modifications to images)
- However, the presented principles are easily extendable to case 2

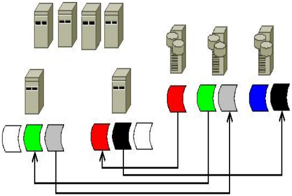
Mirror image contents locally

- Trap reads and writes
- Copy-on-reference (COR), first time reads do:
  - Fetch contents from storage nodes
  - Store contents locally
- Writes and reads on already accessed regions are served locally



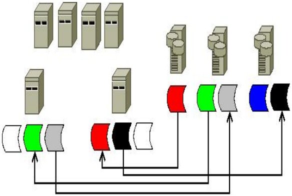
Store initial image contents in a distributed fashion

- Split image into small, equally-sized chunks
- Distribute chunks among storage space providers: distribute I/O workload under concurrency
- Chunk size tradeoff:
  - Too small: long metadata lookup, network overhead
  - Too large: false sharing, large transfers, high latency

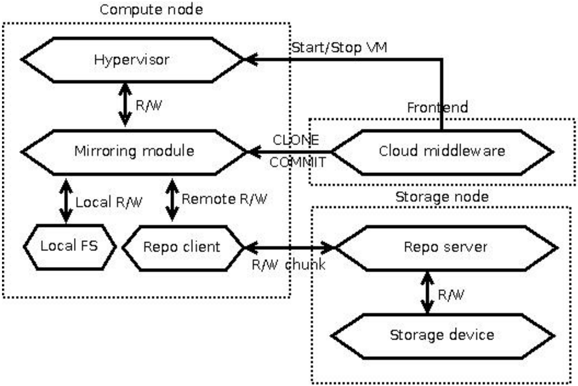


Checkpointing: consolidate local modifications into fully independent images

- Each VM instance has its own local modifications
- On checkpointing, save to repository for each VM local modifications only
- Offer the illusion that a fully independent image is actually stored
- Two control primitives:
  - CLONE
  - COMMIT

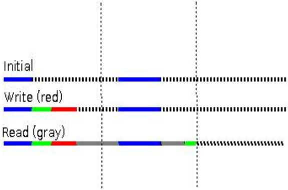


Architecture



Mirroring algorithm

- Considerations
  - Too many small remote reads: low throughput
  - Too many small sparse writes: fragmentation
- Algorithm
  - Hold for each chunk n, such that $(0, n)$ is the largest locally available subsequence
  - On write, fetch remote part to fill the gap and adjust n
  - On read, fetch full chunks only



Advantages

- Fault tolerance
  - A failure on one compute node does not affect other compute nodes
  - Not the case with multicast propagation
- Compatible versioning support
  - Different custom incompatible image formats: AMI, OVF, QCOW2
  - Most hypervisors support RAW format, but it has no versioning support
  - Can consolidate local modifications into fully independent RAW virtual images
  - Enables easy migration from one hypervisor to another

Overview

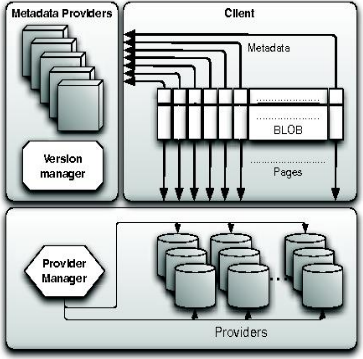
- BlobSeer used to store images and consolidate local modifications
  - Runs on the storage nodes
- Mirroring implemented as a FUSE module
  - Runs on the compute nodes

What is BlobSeer?

- Versioning data storage service for distributed applications
  - Huge BLOBs (order of TB)
  - Fine grain concurrent access (as low as KB order)
  - Cheap versioning and instant cloning
  - High throughput under concurrency
  - Cheap replication and fault tolerance
- Access interface
  - id = CREATE()
  - v = APPEND(id, size, buffer)
  - v = WRITE(id, offset, size, buffer)
  - READ(id, v, offset, size, buffer)
  - (v, size) = GET_RECENT(id)
  - new_id = CLONE(id, v)

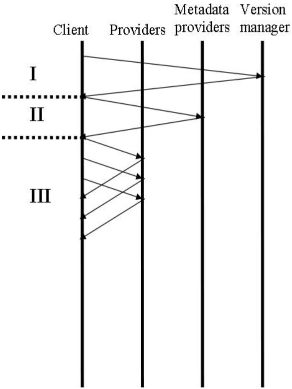
BlobSeer: Architecture

- Principles
  - Data striping
  - Distributed metadata management
  - Versioning based concurrency control
- Actors
  - Data providers
  - Metadata providers
  - Provider manager
    - Allocation strategy
  - Version manager
    - Version assignment and publication



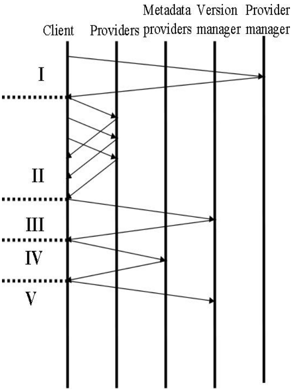
BlobSeer: How does a read work?

1. Optionally ask the version manager for the latest published version
2. Fetch the corresponding metadata from the metadata providers
3. Contact providers in parallel and fetch the chunks into the local buffer



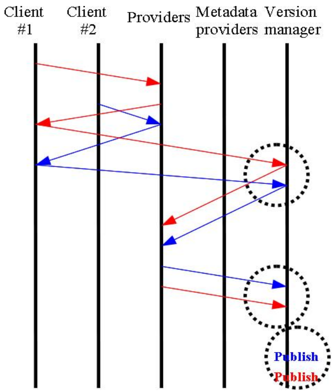
BlobSeer: How does a write work?

1. Get a list of providers that are able to store the pages, one for each page
2. Contact providers in parallel and write the chunks
3. Get a version number for the update
4. Add new metadata to consolidate the new version
5. Report the new version is ready for publication



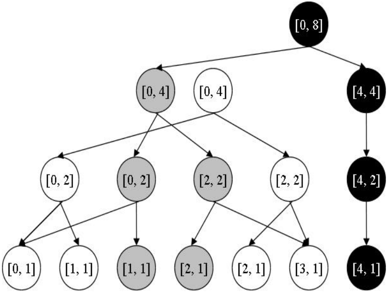
BlobSeer: Concurrent writes

- Chunks are written concurrently by the clients
- Versions are assigned in the order the clients finish writing
- Metadata is written concurrently by the clients
- Versions are published in the order they were assigned



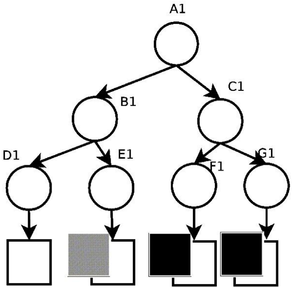
BlobSeer: Zoom on metadata management

- Distributed Segment Tree
  - Each node holds versioning information
  - Write/Append: build nodes up to the root
  - Read: descent from the root towards the leaves



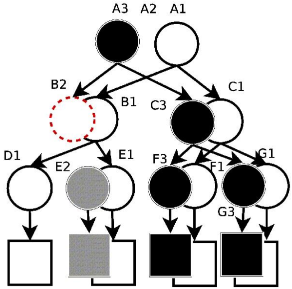
Concurrent writes: Example (1)

- Initial version: white(v_1), two concurrent writers: gray and black
- Gray is first to ask for version number (v_2), black follows (v_3)
- Metadata written concurrently: black is faster, needs to link to B_2
- B_2 does not exist yet, it is a *metadata forward reference*
- Gray finishes writing metadata
- Metadata is now consistent, both v_2 and v_3 are published



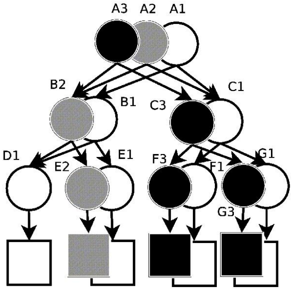
Concurrent writes: Example (2)

- Initial version: white(v_1), two concurrent writers: gray and black
- Gray is first to ask for version number (v_2), black follows (v_3)
- Metadata written concurrently: black is faster, needs to link to B_2
- B_2 does not exist yet, it is a *metadata forward reference*
- Gray finishes writing metadata
- Metadata is now consistent, both v_2 and v_3 are published



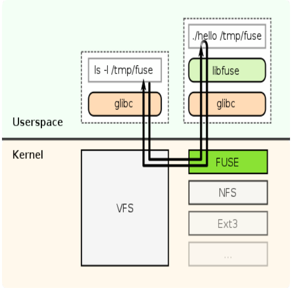
Concurrent writes: Example (3)

- Initial version: white(v_1), two concurrent writers: gray and black
- Gray is first to ask for version number (v_2), black follows (v_3)
- Metadata written concurrently: black is faster, needs to link to B_2
- B_2 does not exist yet, it is a *metadata forward reference*
- Gray finishes writing metadata
- Metadata is now consistent, both v_2 and v_3 are published



FUSE: File System in Userspace

- Bridge from userspace to kernel file system interfaces
- Advantages
  - Create file systems without editing kernel code
  - Access storage devices and services in a standard POSIX fashion
  - Benefit from kernel VFS optimizations
- Disadvantages
  - Context switch overhead



Putting everything together (1)

- Expose BLOBs stored in BlobSeer as regular files through FUSE
  - Two-level namespace: /blob_id/blob_version
- On open, create a sparse local file corresponding to the BLOB
- On read/write apply mirroring algorithm
  - Map local file into RAM using mmap
  - No explicit read/writes to local file
- On close, save internal state (list of modified chunks, etc.)
- On reopen, reload internal state

Putting everything together (2)

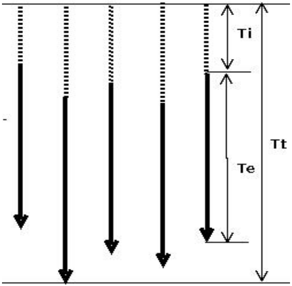
- Implement CLONE and COMMIT as IOCTLs
- CLONE
  - Call BlobSeer CLONE primitive
  - Reassign local file and internal state to new BLOB
- COMMIT
  - Consult list of chunks that have local modifications
  - Group together consecutive modified chunks into a single BlobSeer WRITE
  - Clear list of chunks that have local modifications

Infrastructure

- Grid'5000 experimental testbed
  - 9 sites all over France
  - For this work: Nancy cluster
  - 200 nodes: x86_64, min. 2GB RAM, 120 GB local disk
- Virtual image characteristics
  - Size: 2GB, RAW
  - Distribution: Debian Sid, x86_64

Methodology (1)

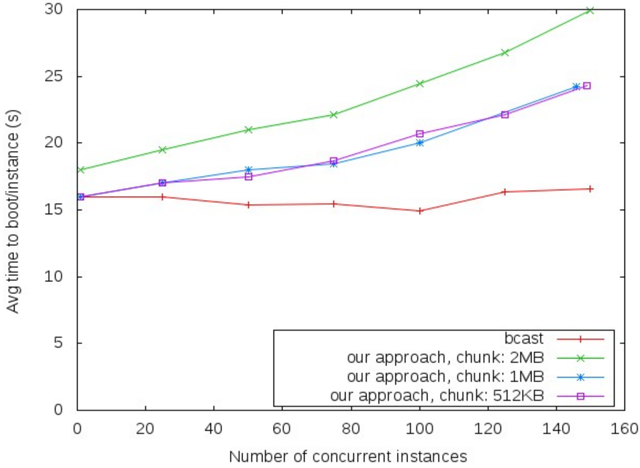
- Simultaneous deployment of VMs
  - 150 compute nodes
  - 50 storage nodes
- VM parameters
  - T_i: Initialization time
  - T_e: Execution time
  - T_t: Total execution time
  - N_t: Total network traffic
- Relevant in our context
  - $AVG(T_e)$
  - T_t
  - N_t



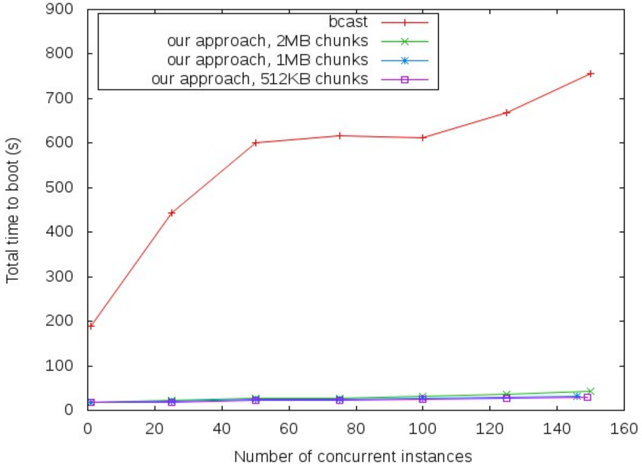
Methodology (2)

- Two approaches evaluated
  - Our approach
    - T_i = time to mount FUSE module locally
  - Full local pre-propagation: TakTuk
    - T_i = time to receive the full copy
    - Propagation through multicast tree
    - Dynamically adjusts tree for optimal latency/bandwidth tradeoff
    - Seeding source: NFS server
- Gradually increase the number of VMs deployed simultaneously

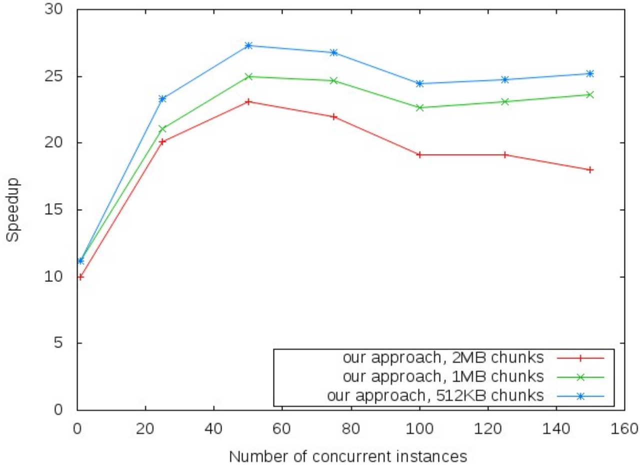
Average time to boot/instance



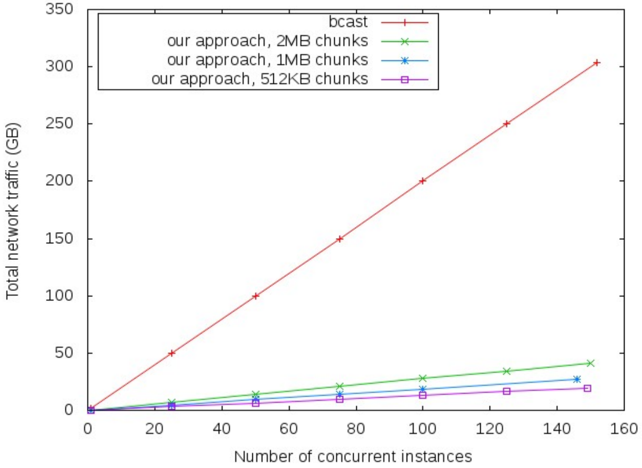
Total time to boot all instances



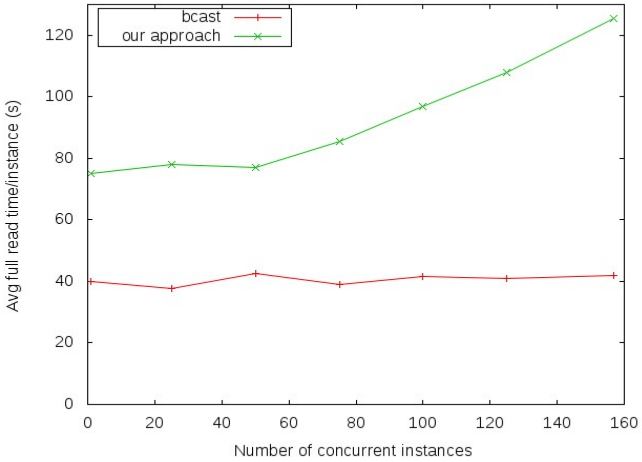
Speedup: boot all instances



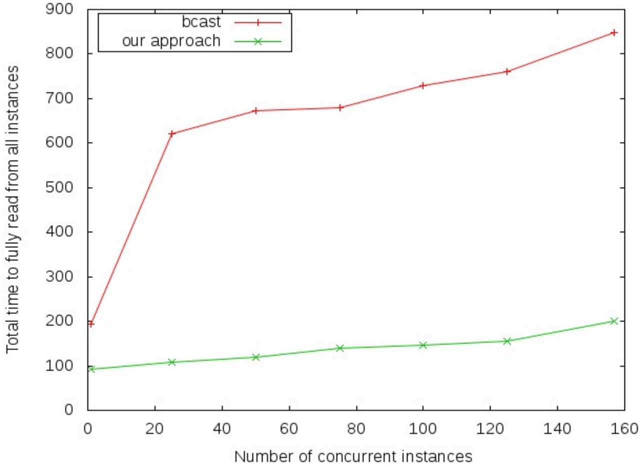
Total network traffic



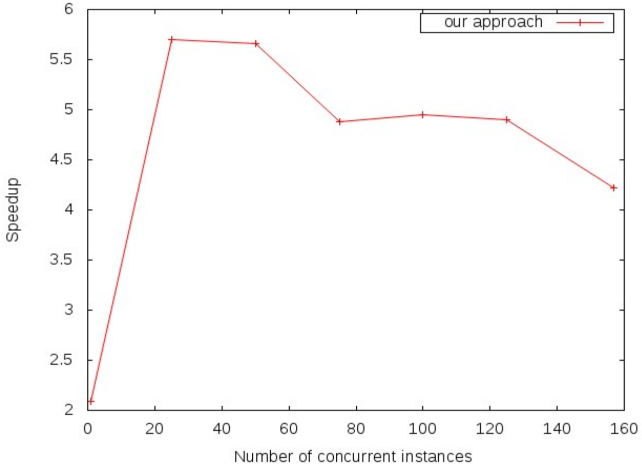
Average boot and full read time/instance



Total time to boot and fully read all instances

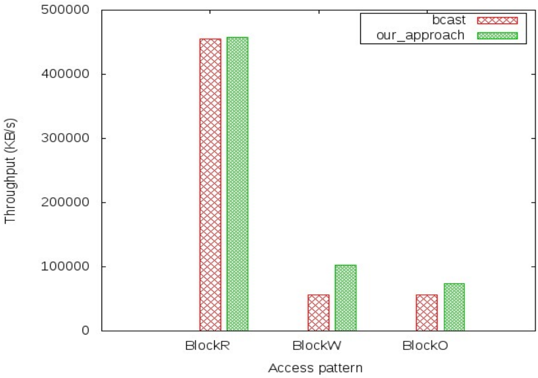


Speedup: boot and fully read all instances

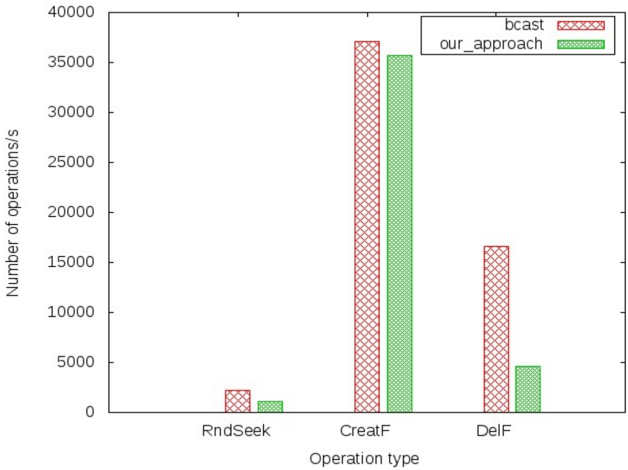


Bonnie++: Throughput

- Single instance, write intensive scenario
- Reads back written data

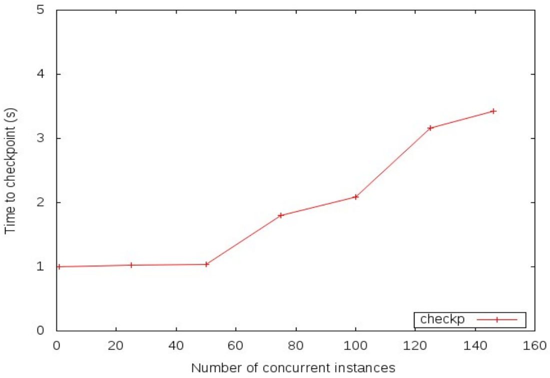


Bonnie++: Number of operations/s

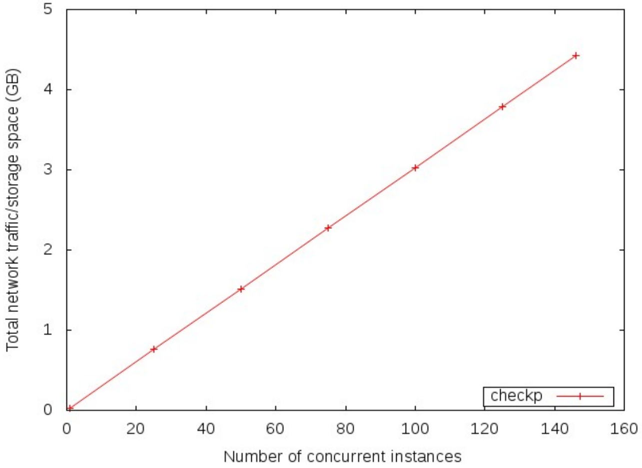


Checkpointing performance: Speed

- 100 instances are booted and a temp file is created (1MB).
- all local modifications are saved simultaneously.



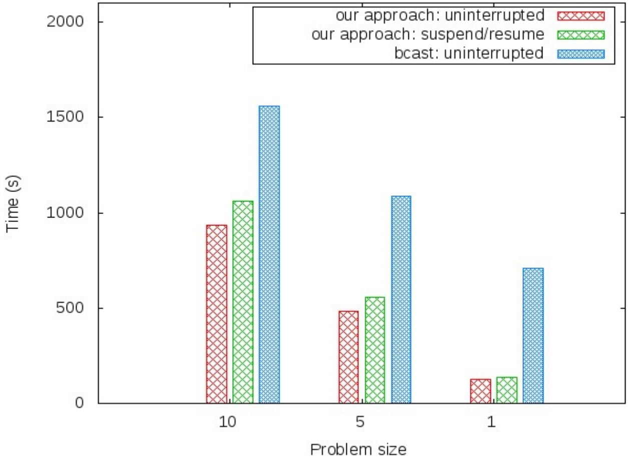
Checkpointing performance: Space



Real application performance: Monte-Carlo PI approx

- Embarassingly parallel problem
- 100 instances, 1/5/10 iterations
- 3 experiments
  - Run uninterrupted using our approach
  - Run using our approach:
    - Checkpoint all, suspend all, resume all after 5mins
    - Each instance resumes on a different node
  - Run uninterrupted after TakTuk propagation

Monte-Carlo PI approx results



Conclusions

- Addressed two difficult challenges
  - VM image content propagation to multiple instances
  - Efficient checkpointing/resume
- Important advantages
  - Fault tolerance
  - Inter-hypervisor compatibility
  - Works with even with no replication
- Encouraging results
  - Boot speedup up to 30x
  - Full read speedup up to 5x
  - Write speedup up to 2x
  - Checkpointing 150 instances takes less than 4s
  - Real life application benefits

Future work

- More experiments
  - Chunk size, Image size
  - Applications
  - LAN vs. WAN
  - Replication
- Access pattern prediction
  - Access pattern feed-back from instances
  - Prefetch according to feed-back
- Integration in Nimbus

Acknowledgements

- Gabriel Antoniu, Luc Bougé: KerData, INRIA/ENS Cachan, Rennes, France
- Diana Moise, Tran Viet-Trung, Alexandra Carpen-Amarie: INRIA/Univ of Rennes 1/ENS Cachan, France
- Jesús Montes, María Pérez, Alberto Sánchez: Univ. Politecnica de Madrid, Spain
- John Breshnahan, Kate Keahey, David LaBissoniere, Tim Freeman: Univ of Chicago/Argonne National Lab, USA
- Matthieu Dorier, Franck Capello, Marc Snir: INRIA/UIUC joint Lab, USA
- Osamu Tatebe: Univ. of Tsukuba, Japan

Research directions

- Cost-effective storage in clouds
  - VMM management
  - Shared virtual storage
  - Quality of service guarantees
  - Dynamically adapt to prices
  - Cheap (i.e. price) versioning: garbage collection, etc.
- High performance storage for HPC
  - Exploit versioning to improve application workflow parallelism
  - High throughput under heavy access concurrency
  - High level I/O access over BlobSeer: MPI-IO, PHDF5, etc.
- Specialized storage for new paradigms
  - MapReduce, Dryad, etc.