Thread-Based Live Checkpointing of Virtual Machines

Vasinee Siripoonya

Kasidit Chanchio

Department of Computer Science

Thammasat University

THAILAND

Outline

- Introduction
- Motivations
- Backgrounds
- Design and Implementation
- Experimental Results
- Conclusion and Future Works

Introduction (1)

* Virtualization is the enabling technology of Cloud Computing
* Despites many advantages, the Cloud needs fault-tolerance (FT)
* Checkpointing is a common FT technique
  - Desirable for long-running applications
* Traditional Checkpointing Mechanisms are <u>not transparent</u> to users. Users have to:
  - Modify source code
  - Install FT middle-ware runtime system
  - Use a particular OS kernel. etc

Introduction (2)

* VM checkpointing is transparent but hard because VM instances could be large
* Examples: Amazon VM instance
  - Large: 2 vcpu and 7.5 GB RAM
  - Extra large: 4 vcpu and 15 GB RAM
  - Other: 8 vcpus with 17.1 GB RAM,
    8 vcpus with 34.2 GB RAM,
    8 vcpus with 68.4 GB RAM
* How to efficiently chkpt and migrate them?
  - Existing solutions either cause long disruption of services or long chkpt latency (discuss in the background section)

Motivations

* We believe FT should be transparent to users, applications, and Operating Systems.
* Hypervisors already have capabilities to save, restore, and migrate VM state
  - we can leverage them (reuse components)
* Efficient checkpointing capability can provide a new service level to the Cloud
* Powerful computing resources are available
  - servers are multi-cores
  - reduction of memory prices, availability of SSD

Proposed Solution

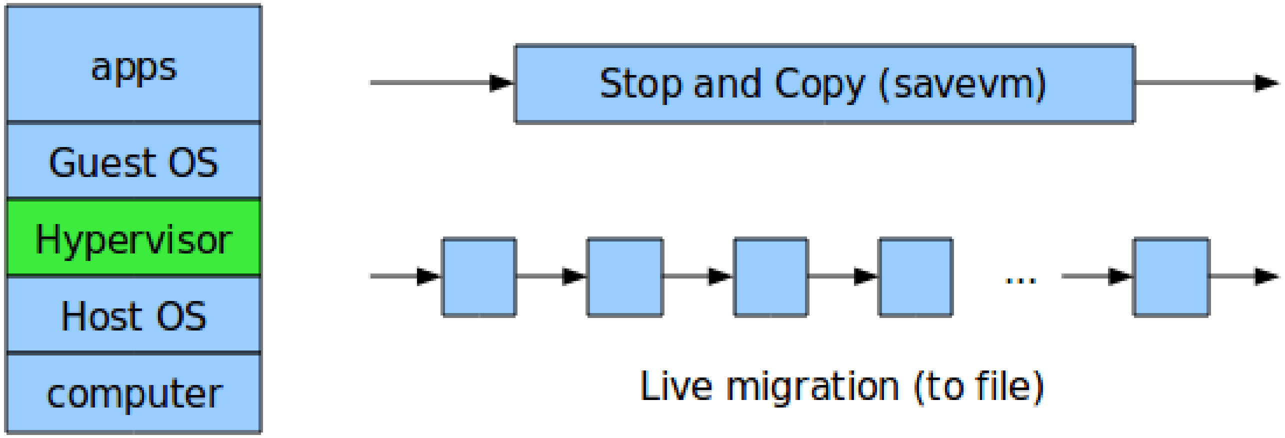
* We propose the Thread-based Live Checkpointing (TLC) mechanism
  - Perform at the hypervisor level
* We define Live Checkpointing as the ability to perform checkpointing operations while allowing computation to progress
* The Checkpointing Thread is introduced to handle most checkpointing tasks

Contributions

* Develop a Transparent and Efficient VM Checkpointing Mechanism
  - TLC can hide Chkpt latency and reduce Chkpt overheads on VM with mem-intensive workloads
* Allow the execution and I/O operations of the VM to progress live during Checkpointing
  - Maintain responsiveness during Chkpting
* Finish the Checkpointing Operations within a bounded period of time

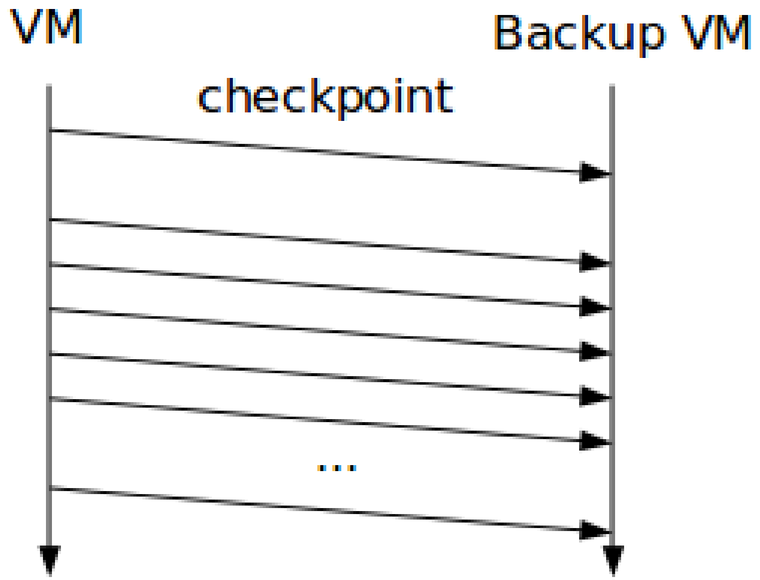
Backgrounds (1)

* Leverage existing mechanisms
  - VM state saving/loading: Long disruption of services
  - Use live and non-live “migration to file”: Checkpoint latency is too long



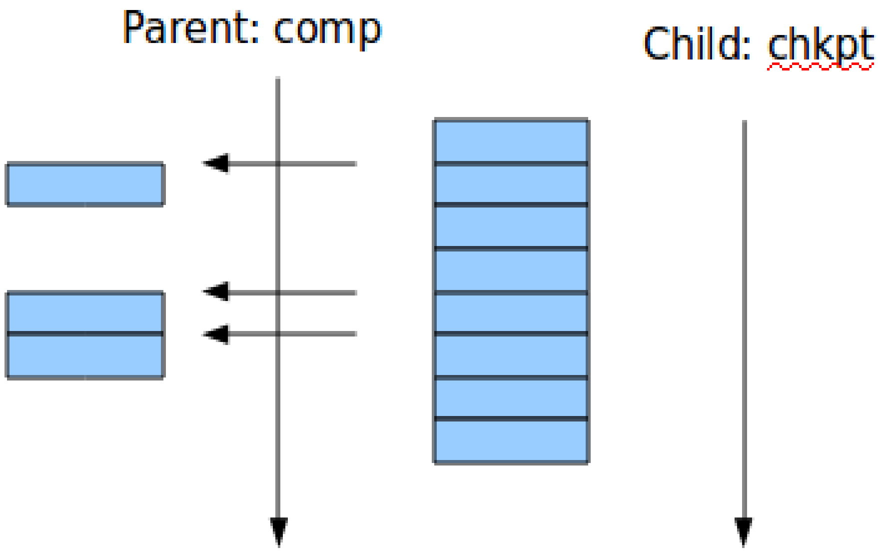
Backgrounds (2)

* VM Fail-Over e.g. REMUS & kermari
  - Incrementally sync state to backup VM
  - Overkill for chkpting



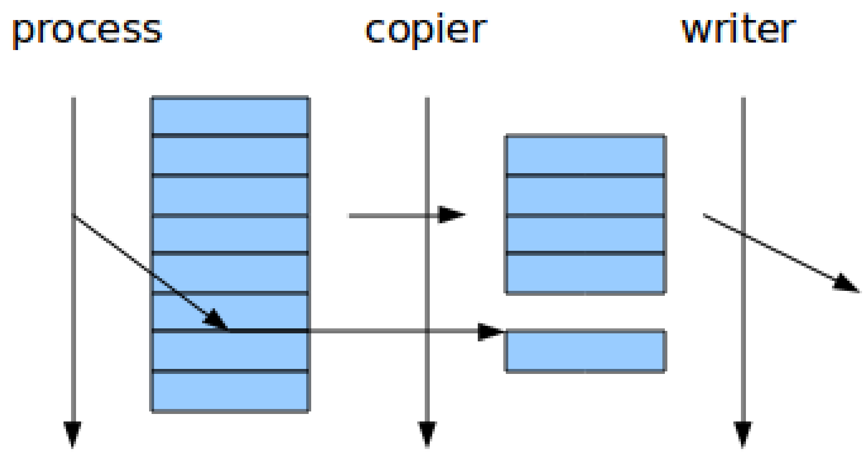
Backgrounds (3)

* Copy-On-Write mechanisms of Libckpt (J. Plank)
  - Similar to fork
  - Extensive ram updates can also cause long disruption of services

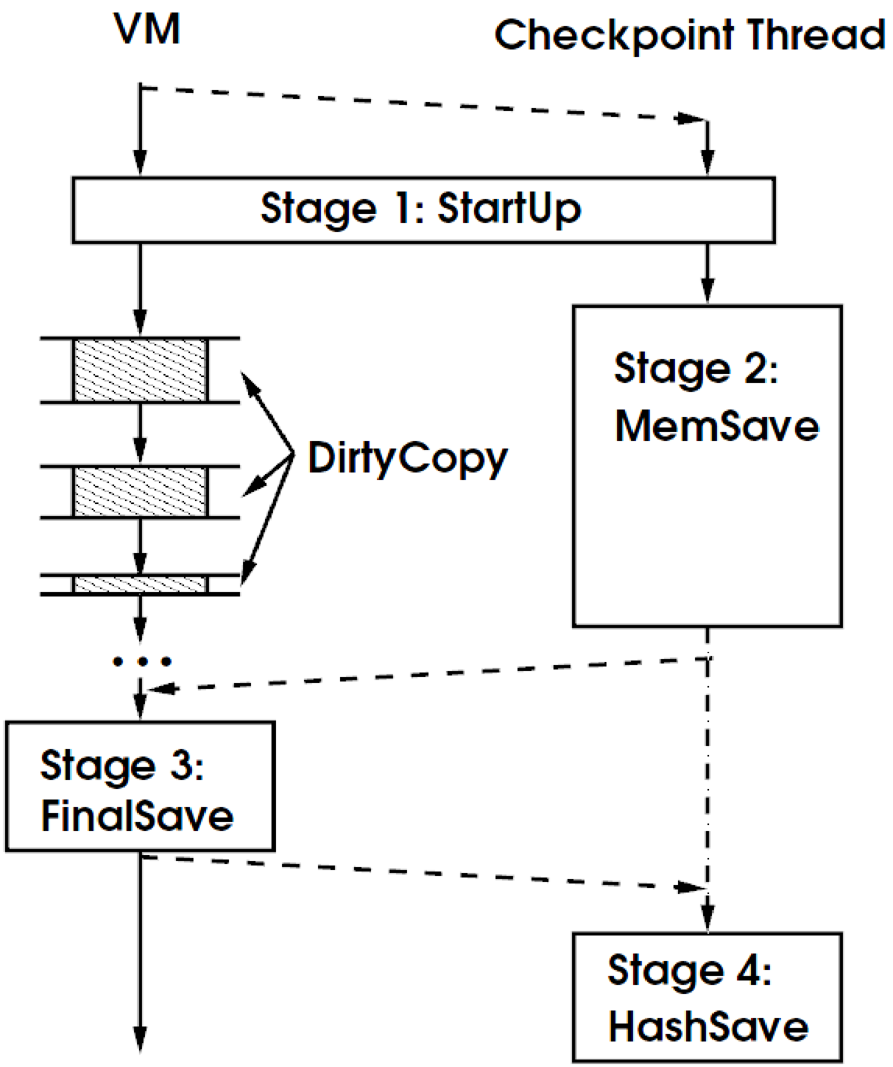


Backgrounds (4)

* Concurrent Checkpointing (K. Li, J. Naughton, and J. Plank)
  - Employ copier and writer threads
  - A mem page is copied to a buffer on every mem write during checkpointing



TLC Design


VM
Checkpoint Thread
Stage 1: StartUp
Stage 2: MemSave
DirtyCopy
Stage 3: FinalSave
Stage 4: HashSave

Correctness

* The Chkpt thread captures the VM state *at the Chkpt moment* while the "dirty copy" interrupt handler takes care of dirty pages generated *beyond that point*
* Hash Table (HT) is used to keep dirty pages (which could grow large)
* **Correctness:** The mem pages copied by the Chkpt thread could be corrupted due to concurrent updates *but contents in the HT and Stage 3 will always correct them*

Optimization/Interruption

* There are opportunities for optimizations at various Stages of this model
  - Opt 1: Chkpt thread only saves pages that have not been copied to HT
  - Opt 2: Stage 4 excludes pages in Stage 3
* The interrupt handler periodically disrupts the VM computation and/or IO operations *<u>to various degrees depending on the architectures of the hypervisor.</u>*

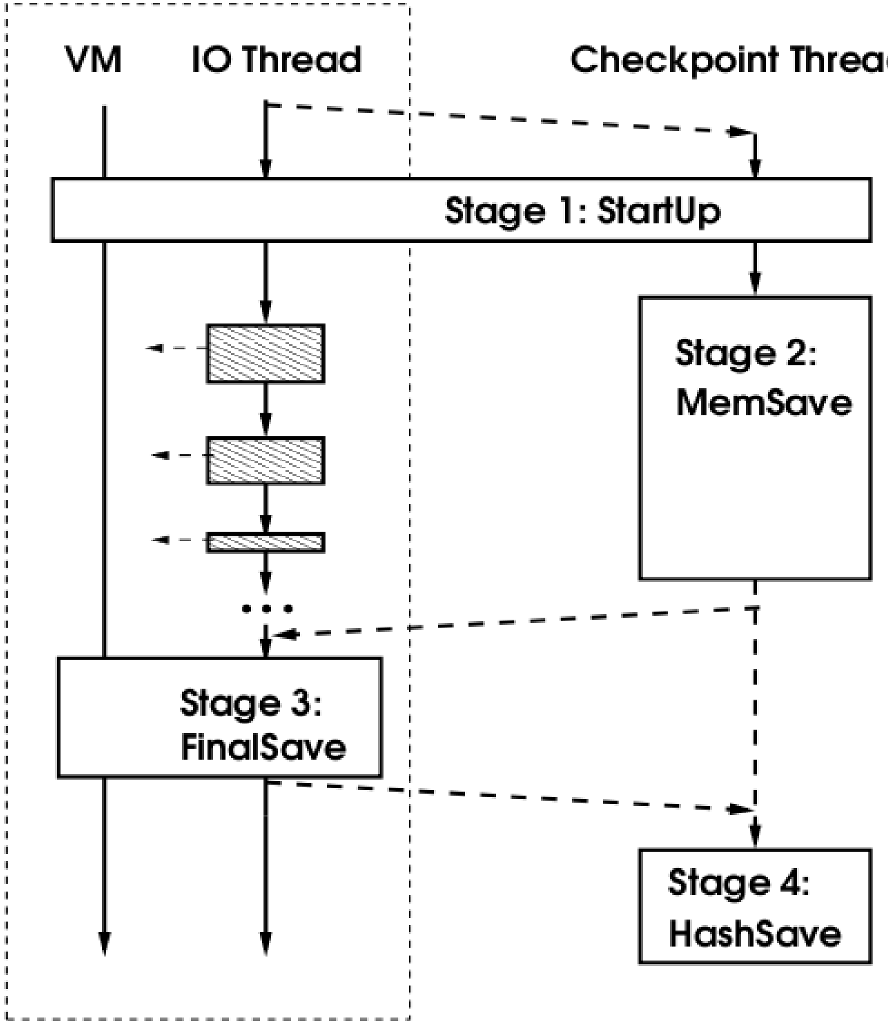
VM Disk Image State

* TLC create a disk image snapshot by creating a (QCOW) overlay disk on top of the current one
* The snapshot capability of “Btrfs” and “ZFS” can also be used for this

TLC Implementation (1)

* We leverage KVM live migration code
* Original KVM architecture uses a single thread execution to
  - compute, do I/O, perform VM services
* Current KVM architecture uses
  - KVM thread to execute vm (1 per vcpu)
  - I/O Thread for I/O ops and VM services

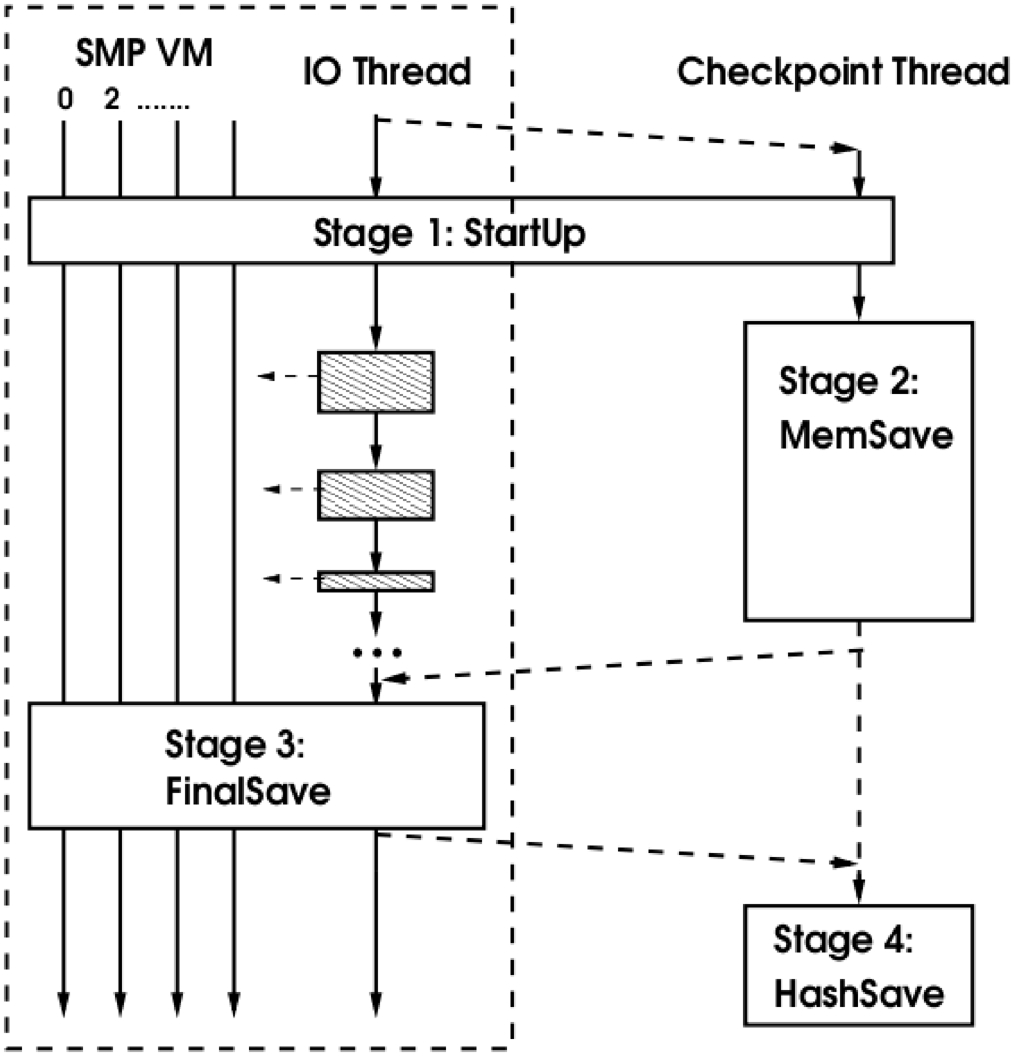
TLC Implementation (2)



vasabiLab
Virtualization Architecture and ScalABle Infrastructure Laboratory

DEPARTMENT OF COMPUTER SCIENCE THAMMASAT UNIVERSITY

TLC Implementation (SMP)

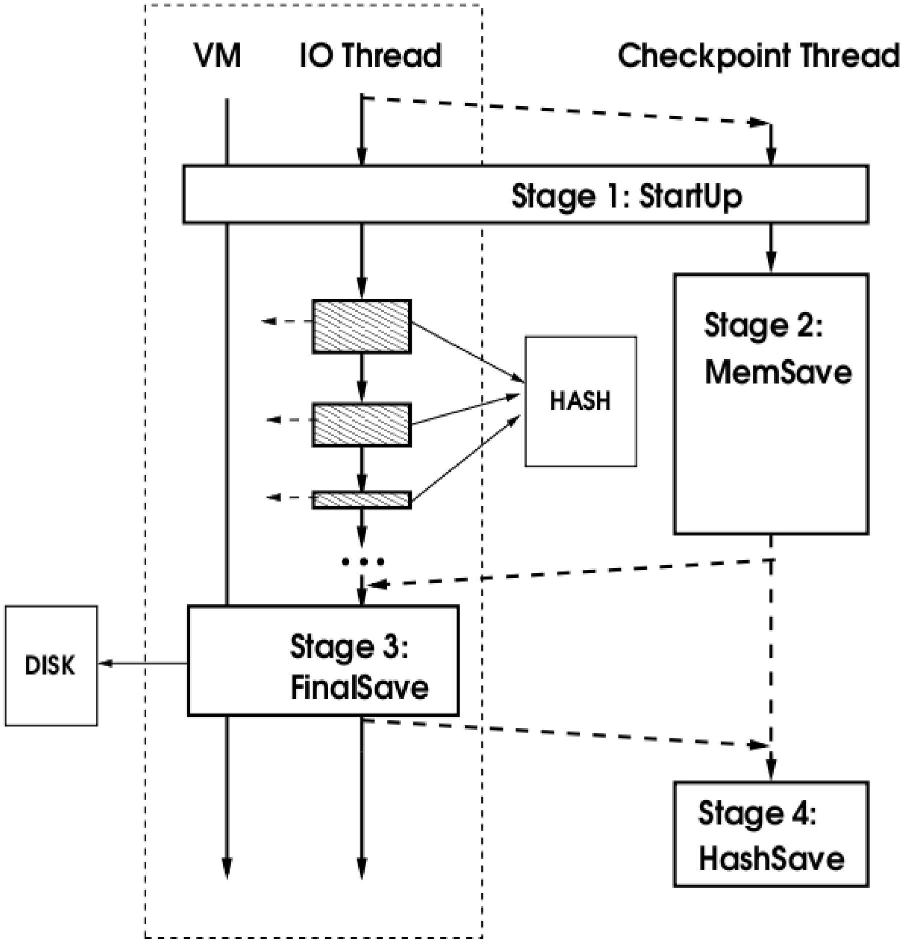


Experiments

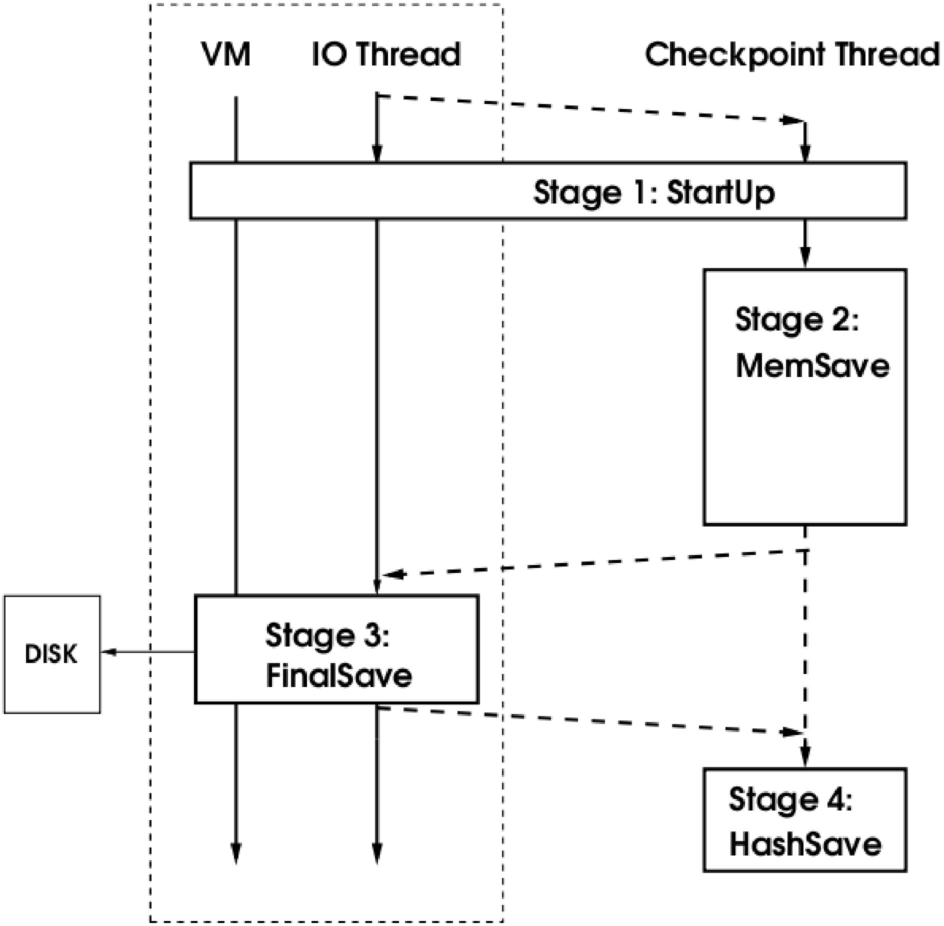
* Experiments on 1 vcpu
  - Evaluate TLC performance
  - Evaluate TLC responsiveness
* Host: two Quad-core Xeon 5500 2.4Ghz 72GB with RAID 0 (256 disk cache) 250GB SATA running Linux 2.6.32 + X11
* Guest: a Fedora 11 Linux 2.6.32 + X11 RAM varied by Workloads (NPB benchmarks)
  – EP.B (345MB, 512MB), MG.B (745MB, 1GB)
  – IS.C (1.3GB, 2GB), FT.B (1.6GB, 2GB)

TLC configurations

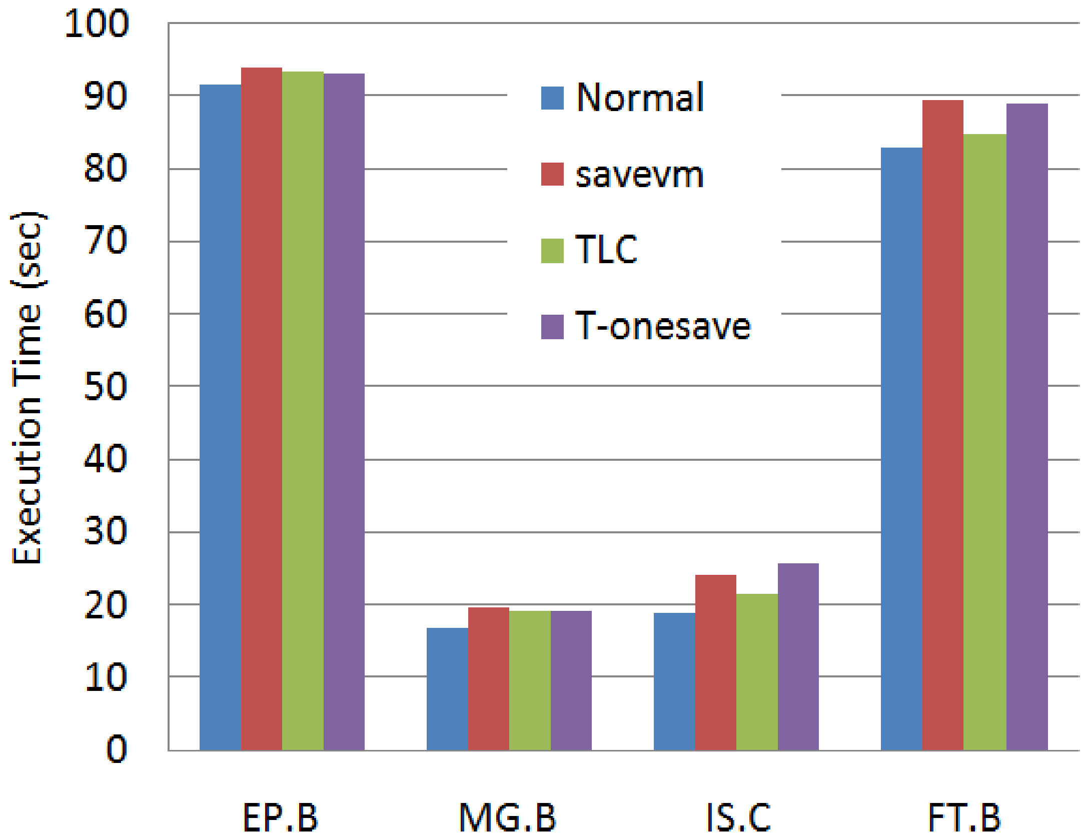
TLC



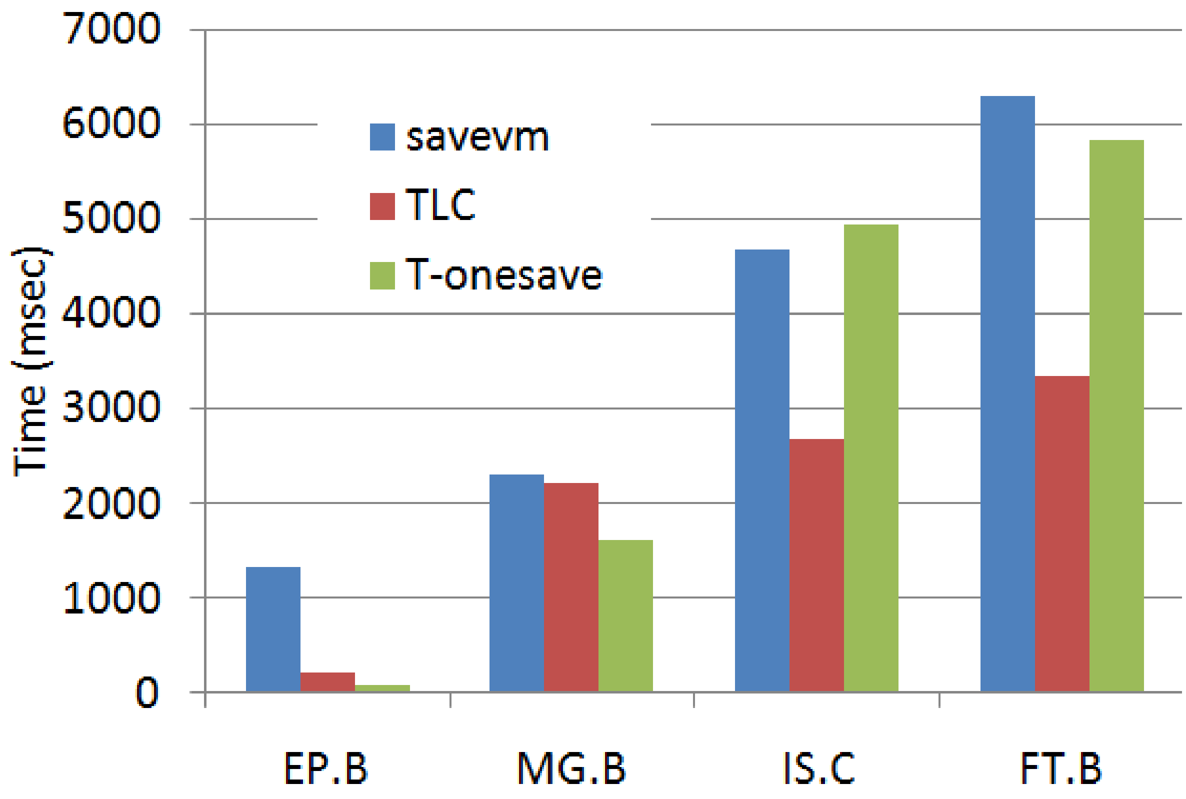
TLC onepass



Execution Time

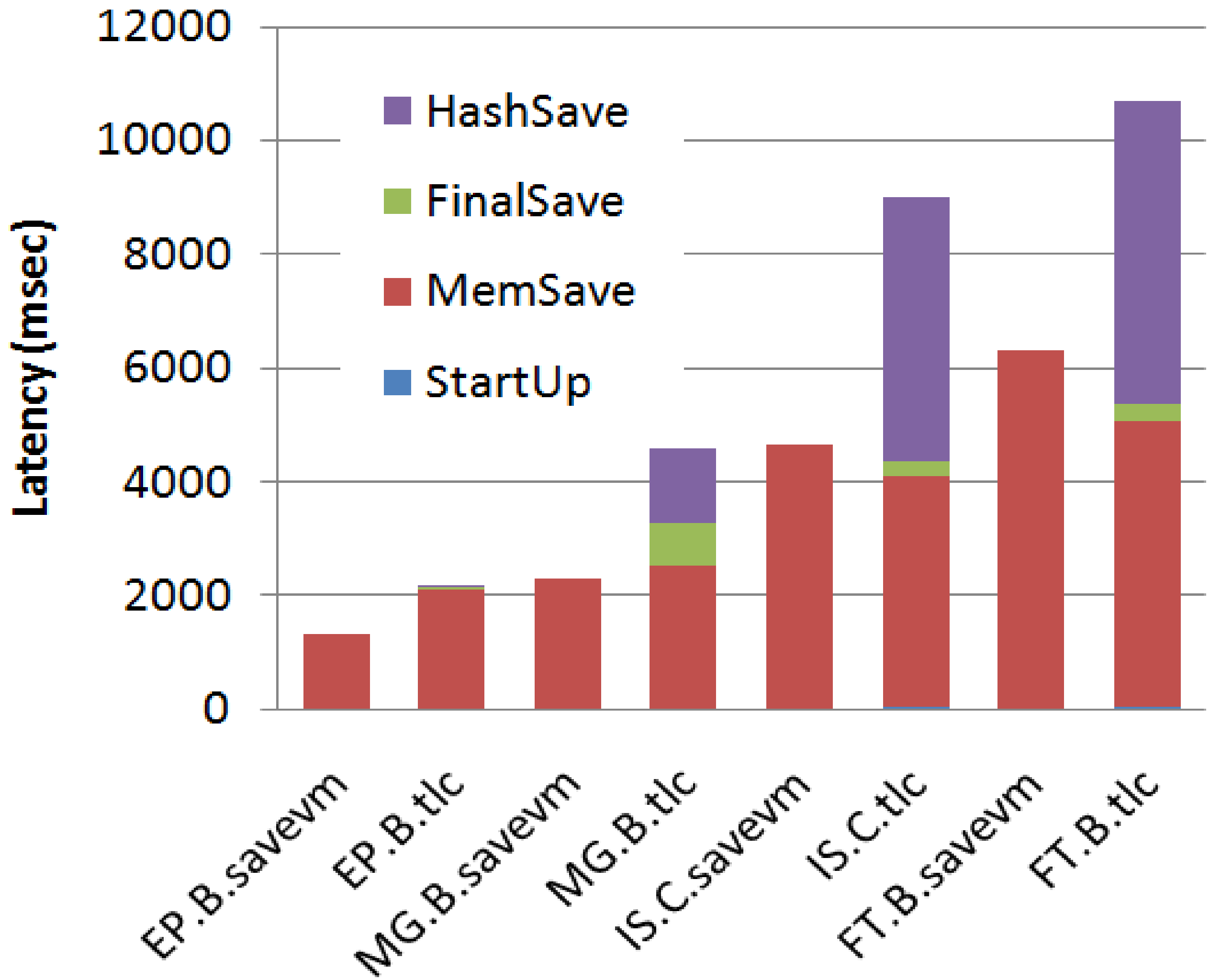


Overheads



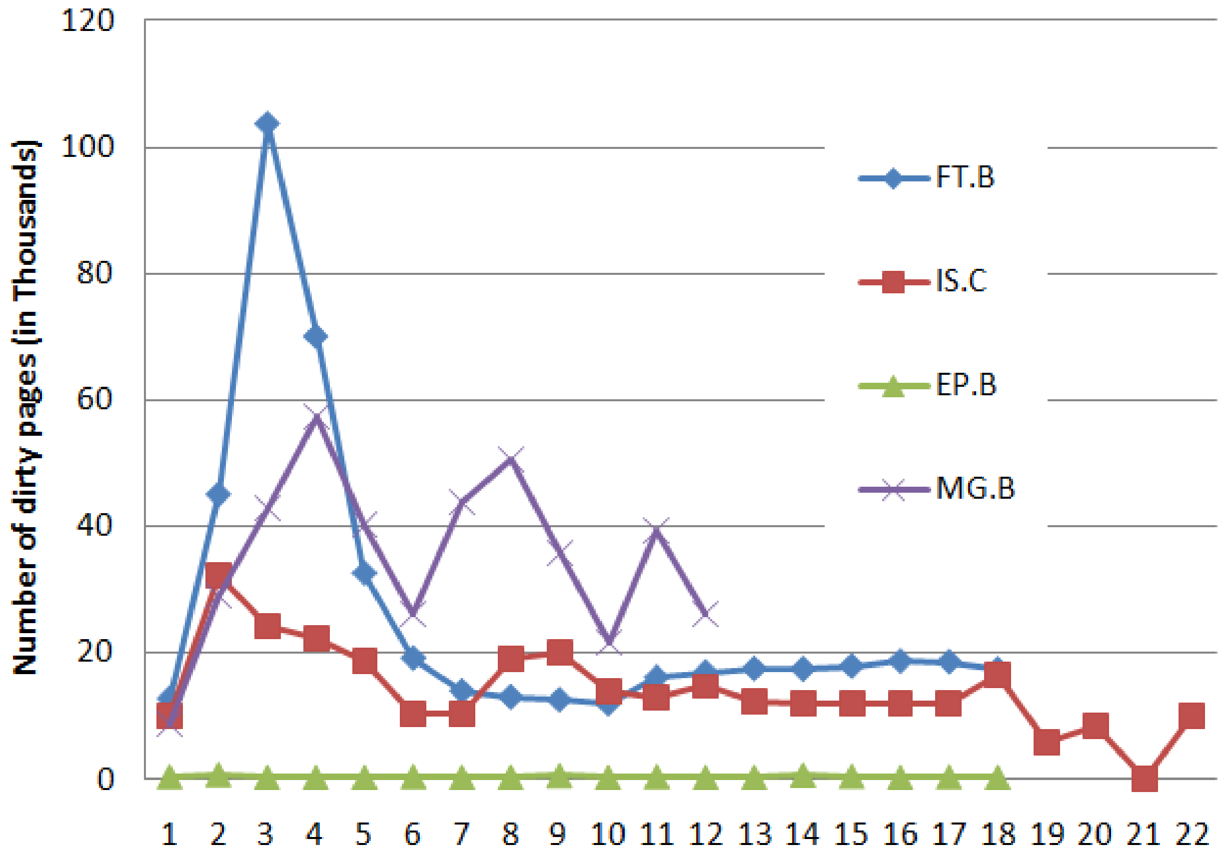
* amount of time increase on VM execution due to checkpointing

Latency



Latency (msec)
12000
10000
8000
6000
4000
2000
0
HashSave
FinalSave
MemSave
StartUp
EP.B.savevm
EP.B.tlc
MG.B.savevm
MG.B.tlc
IS.C.savevm
IS.C.tlc
FT.B.savevm
FT.B.tlc

Dirty Page Copying



Memory Requirements

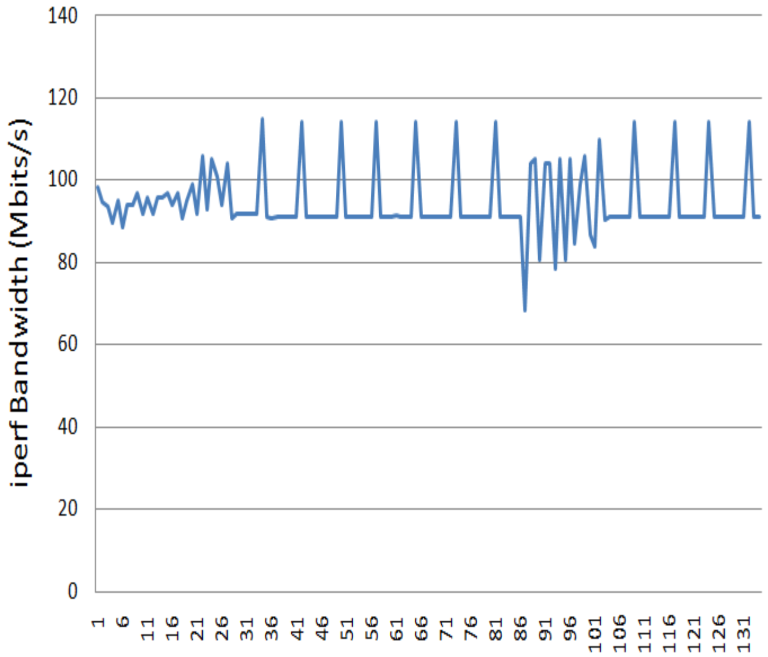
TABLE I. THE AVERAGE NUMBER OF PAGES REQUIRED BY TLC.

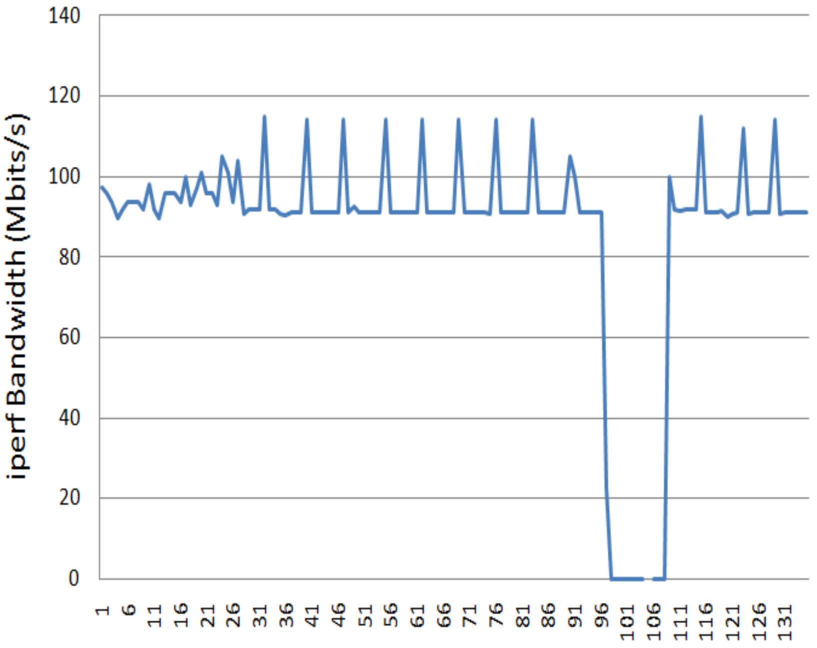
	EP.B	MG.B	IS.C	FT.B
VM memory	133184	264256	526400	526400
Working Set	108289.1	237016.1	400195.9	462750.3
Hash Table	1691.7	112459.5	273753.9	325022.3
Hash/Work Set	0.015	0.47	0.68	0.70

TABLE II. TABLE SHOWING TLC OPTIMIZATION PERFORMANCE.

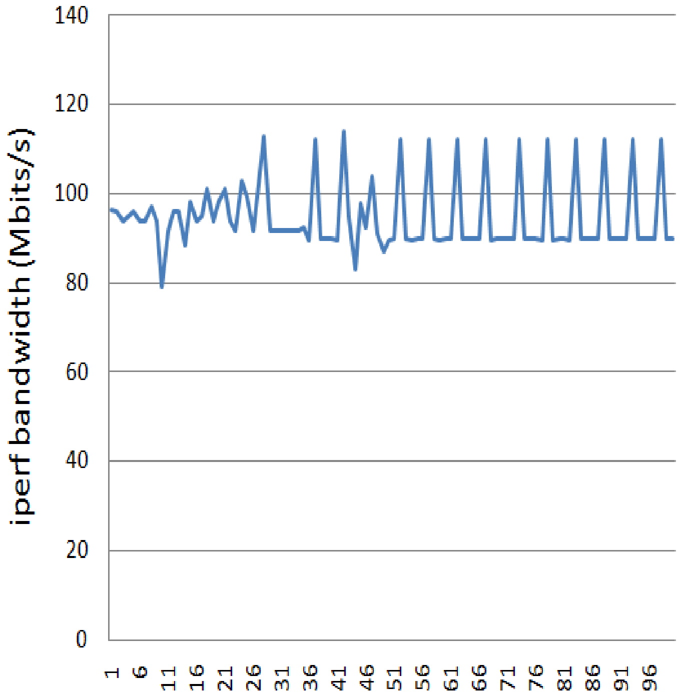
	EP.B	MG.B	IS.C	FT.B
Opt1 Skipped	1173	64853.2	191076.5	169797.9
Opt2 Skipped	307.6	38085.7	11761.8	14445.6
Saved Pages	108500.2	246536.7	471111.5	603529.1
Skipped/Saved	0.01	0.41	0.43	0.30

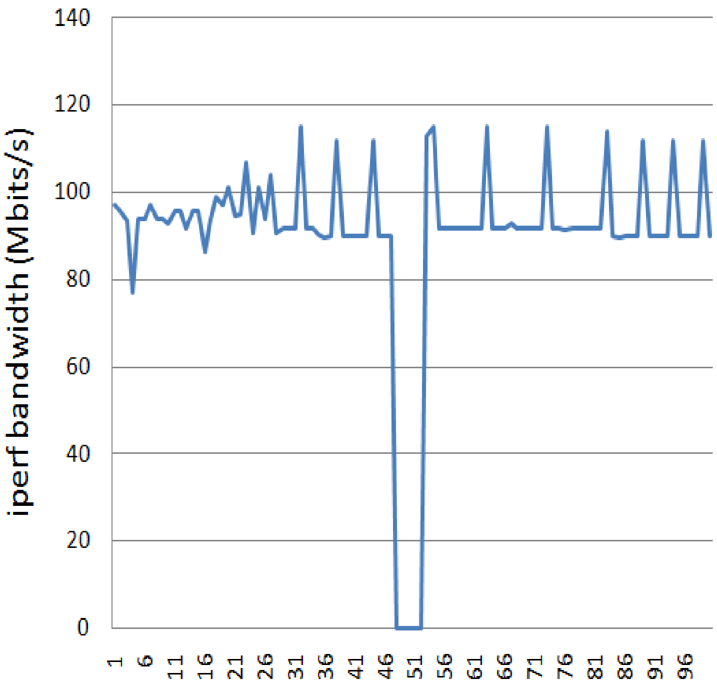
Iperf Performance on FT.C





Iperf Performance on Mg.B

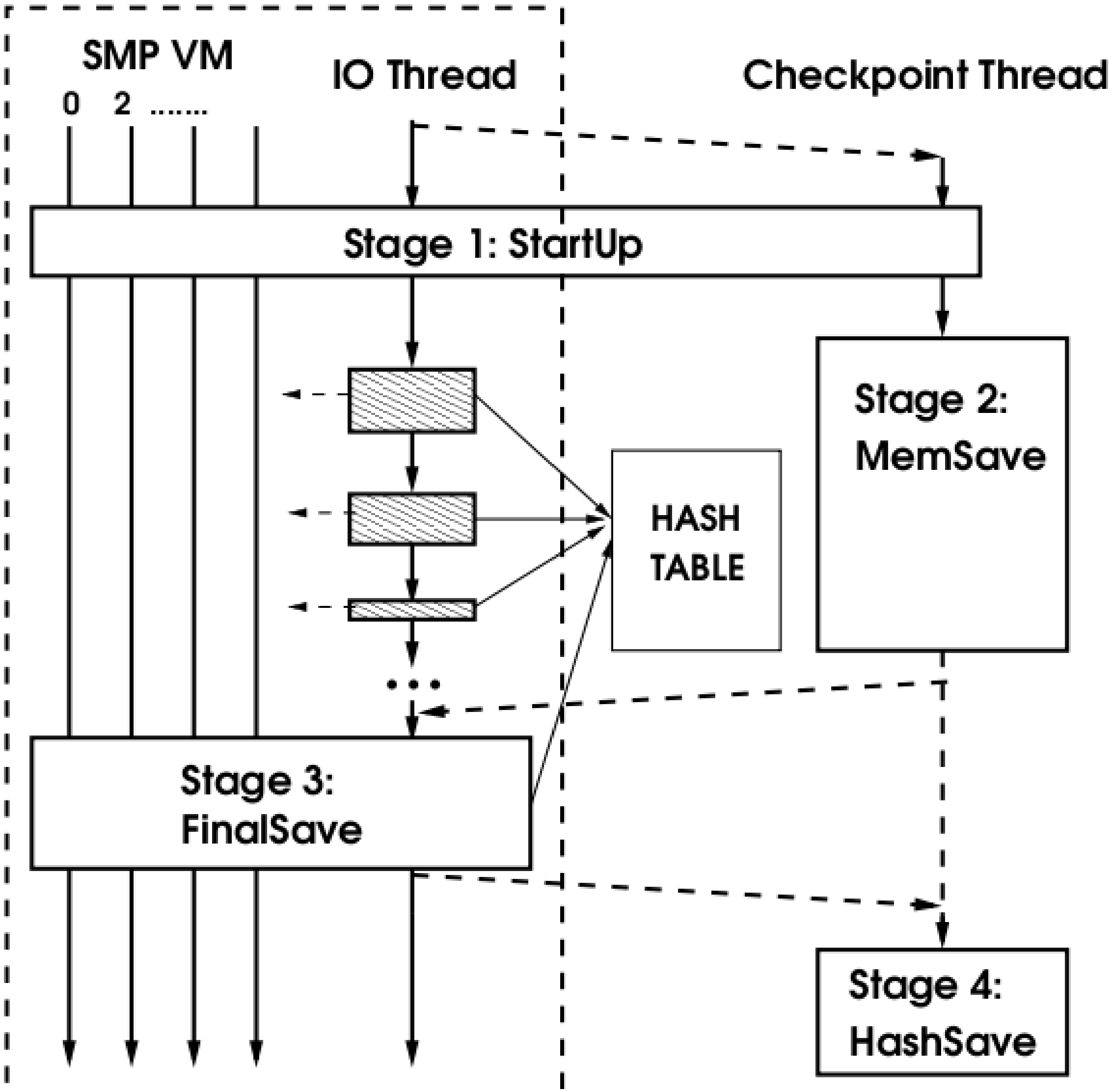




TLC on SMP

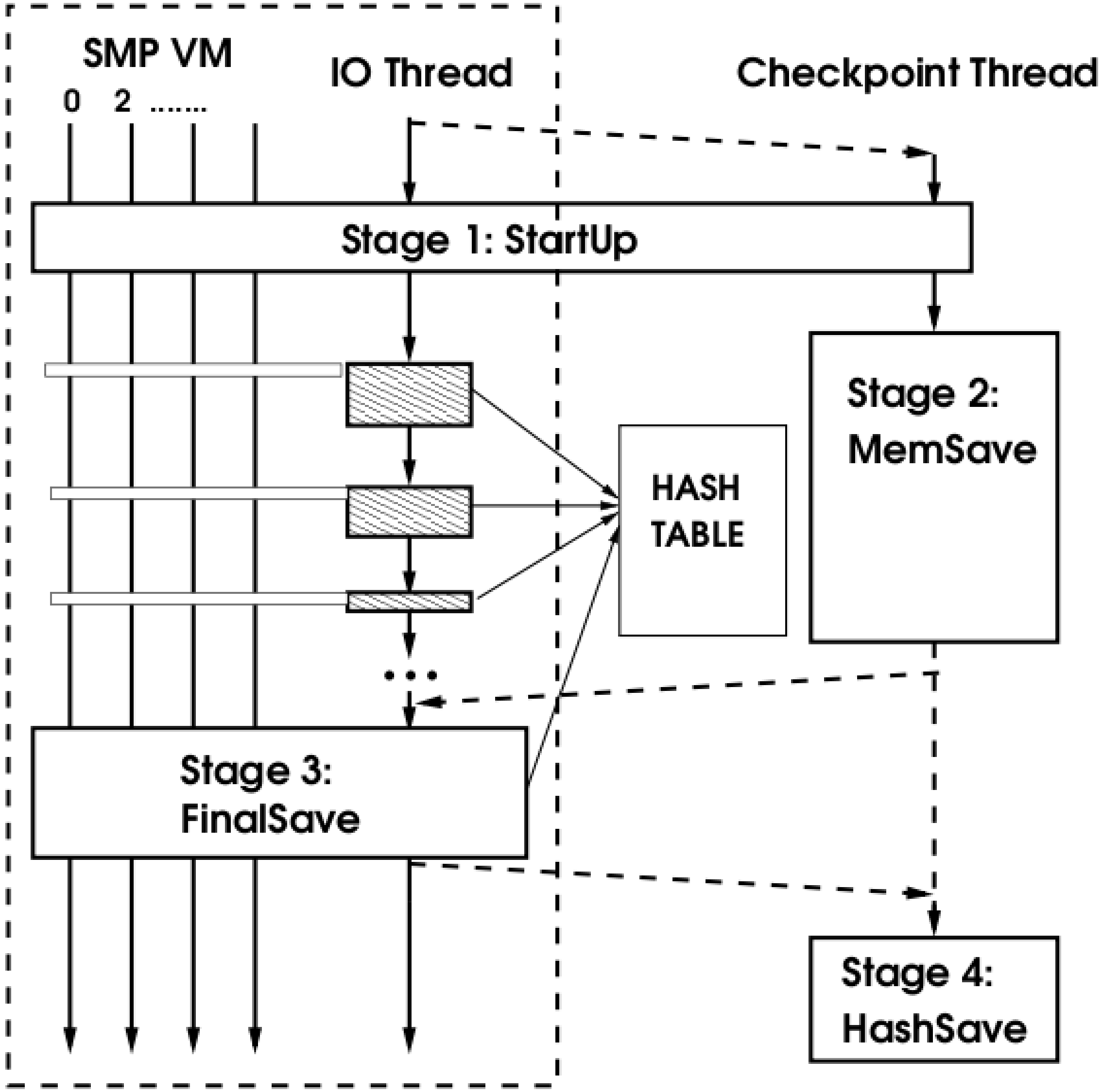
* Host: two Quad-core Xeon 5500 2.4Ghz 72GB with RAID 0 (256 disk cache) 250GB SATA running Linux 2.6.32 + X11
* Guest: SMP 6 vcpu 32GB RAM running Linux 2.6.38 + X11
* We test TLC with mg.D benchmark (using about 86% of Guest Ram)

TLC (nonstop)



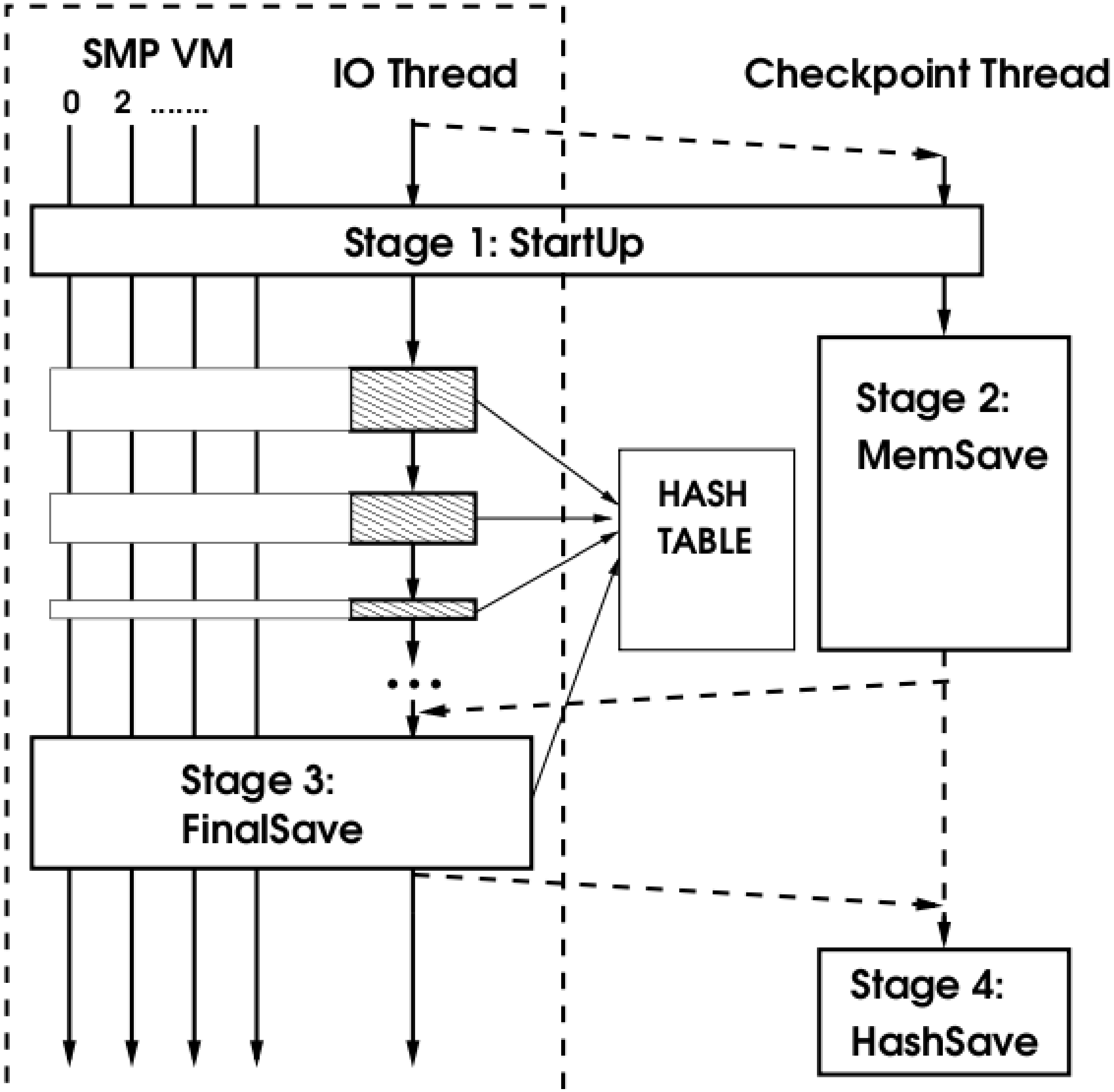
SMP VM
0 2 ....
IO Thread
Checkpoint Thread
Stage 1: StartUp
Stage 2: MemSave
HASH TABLE
Stage 3: FinalSave
Stage 4: HashSave

TLC SMP (log)



DEPARTMENT OF
COMPUTER SCIENCE
THAMMASAT UNIVERSITY

TLC SMP (mem)

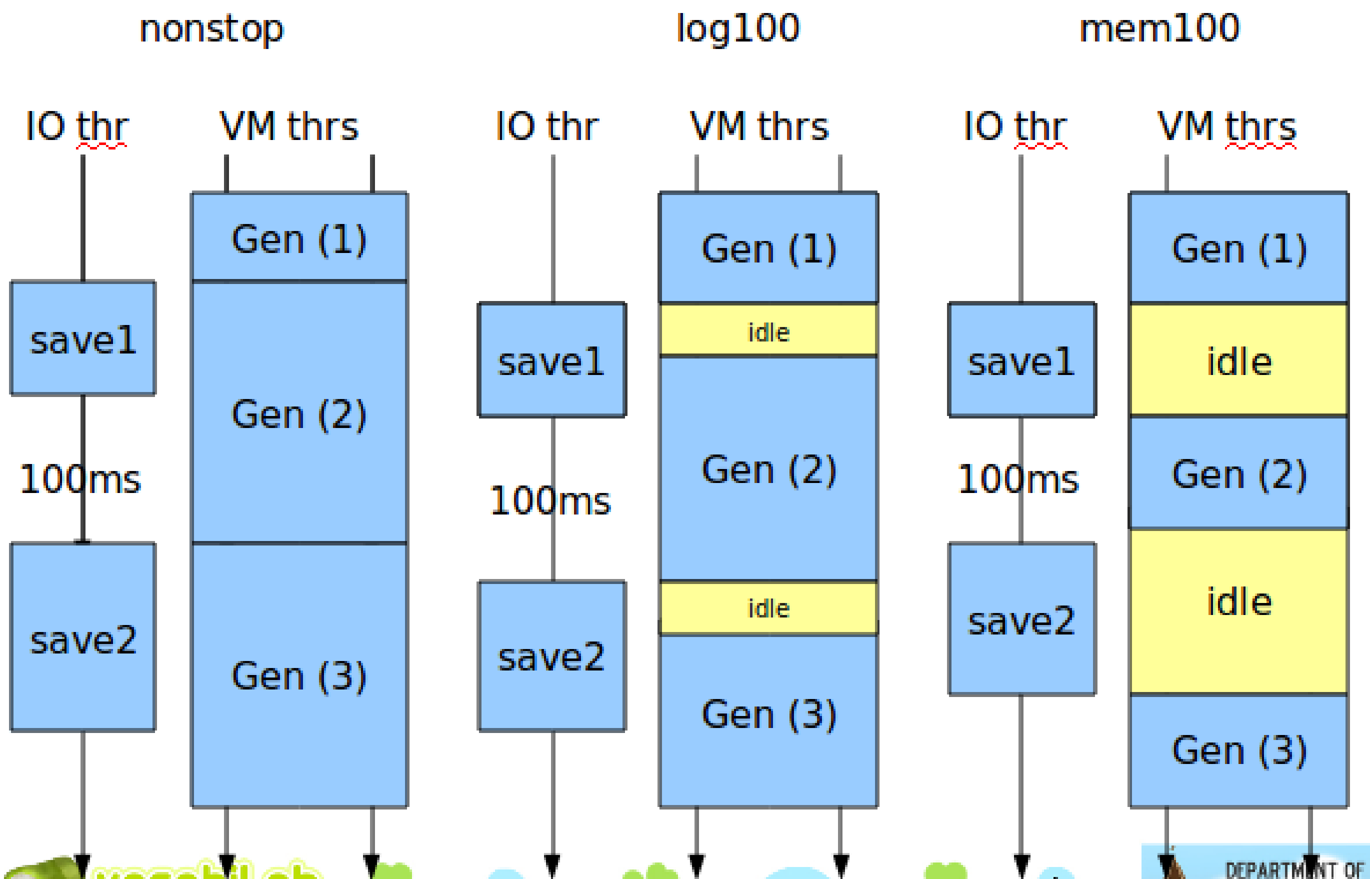


vasabiLab
Virtualization Architec
ScalABle Infrastructure La
DEPARTMENT OF
COMPUTER SCIENCE
THAMMASAT UNIVERSITY

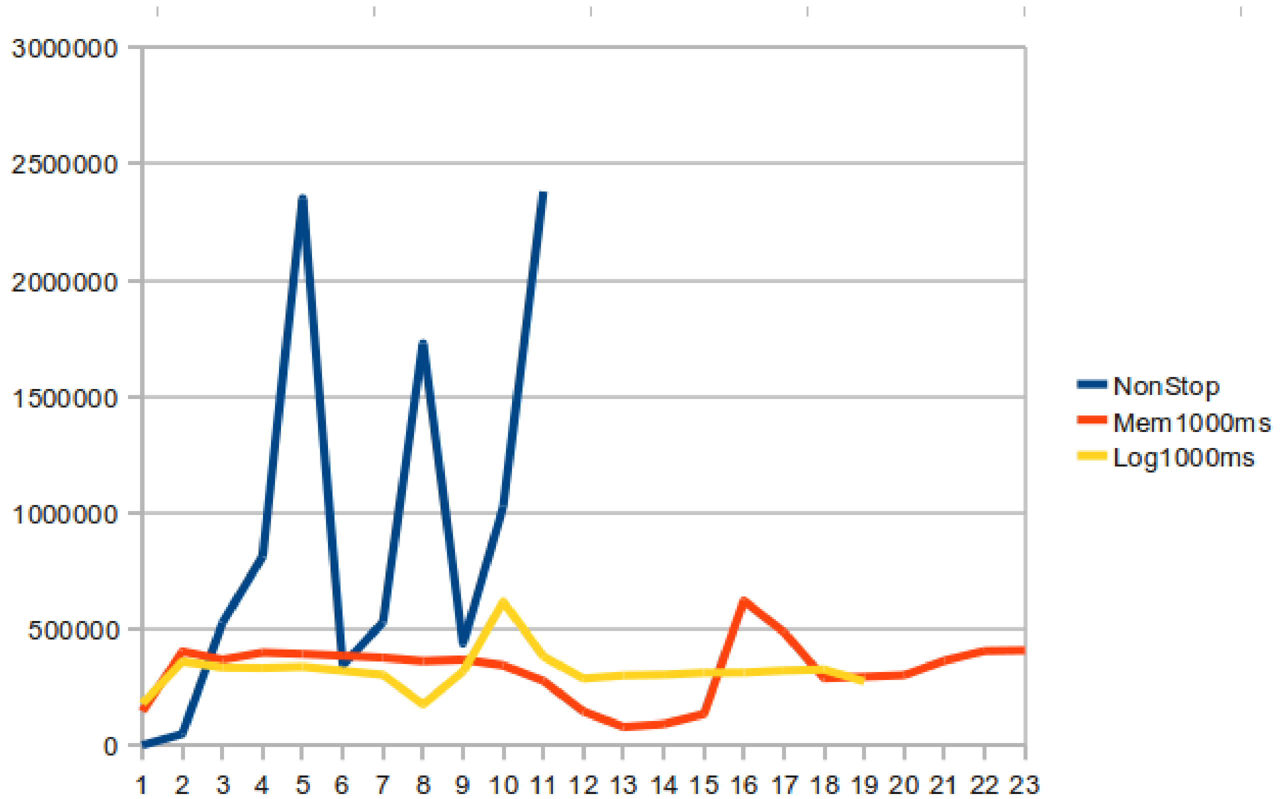
TLC SMP Preliminary Results

* savevm: 104.9 s
* TLC hash size: 20 – 28GB
* Overhead:
  - tlc (nonstop): 76 s
  - tlc (log1000): 60 s
  - tlc (mem1000): 58 s
  - tlc (onepass): 54 s
* Latency:
  - tlc (nonstop): 350 s
  - tlc (log1000): 347 s
  - tlc (mem1000): 377 s
  - tlc (onepass): 585 s

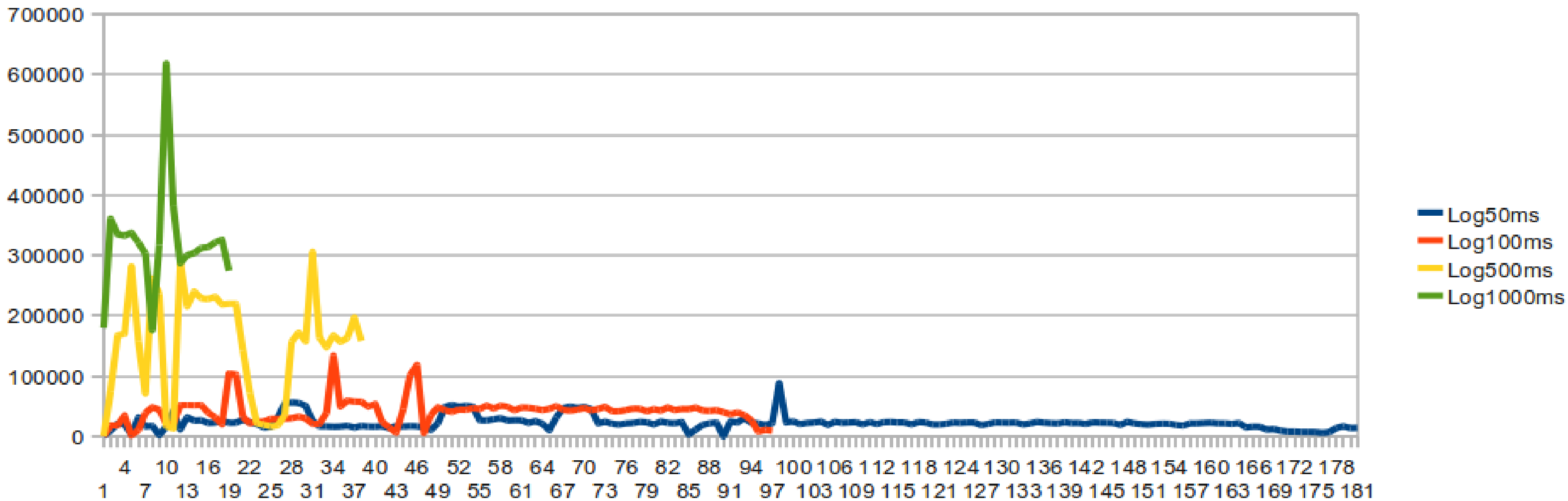
Dirty Page Copying (1)

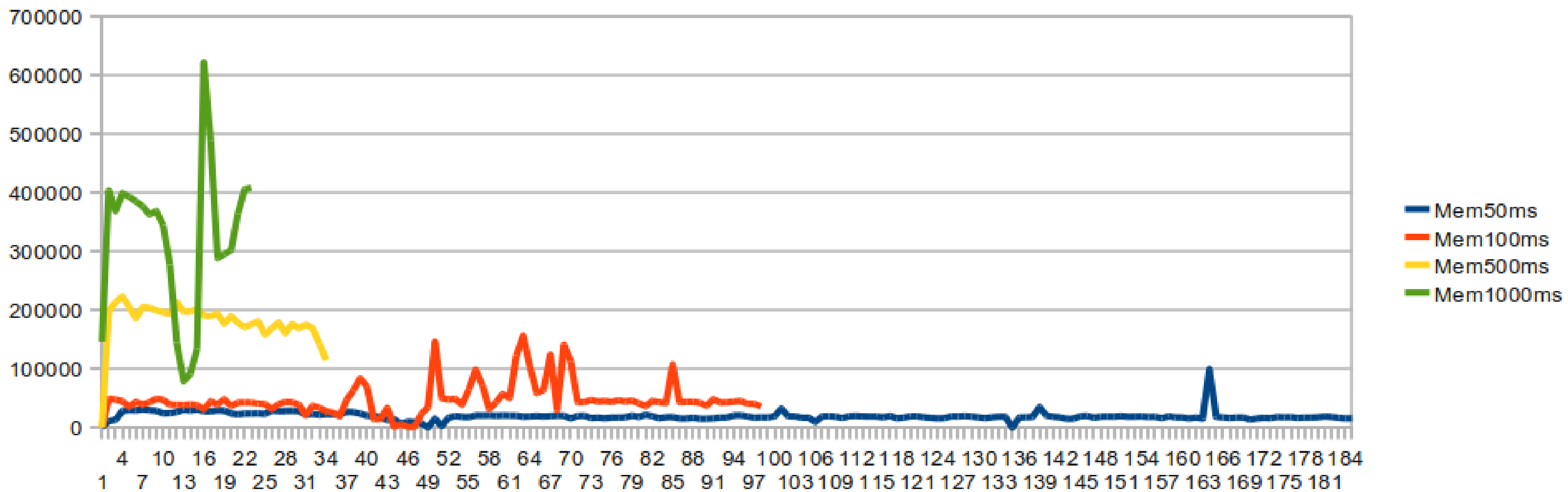


Dirty Page Copying (2)



Dirty Page Copying (3)

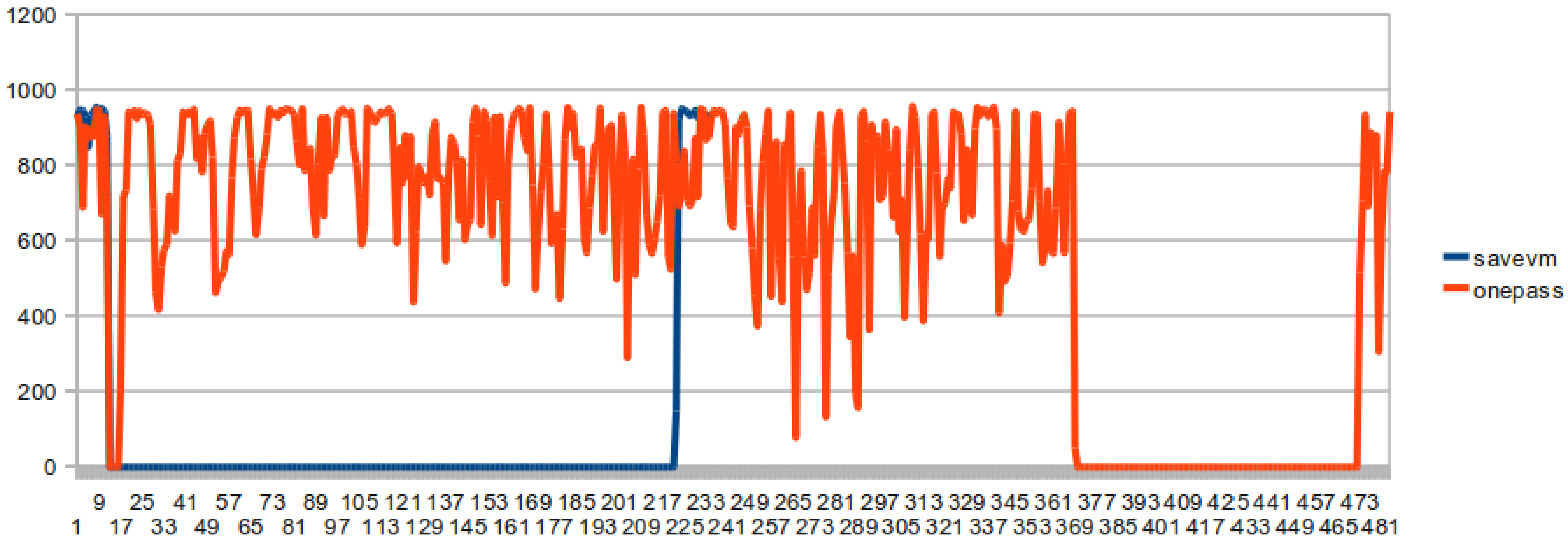




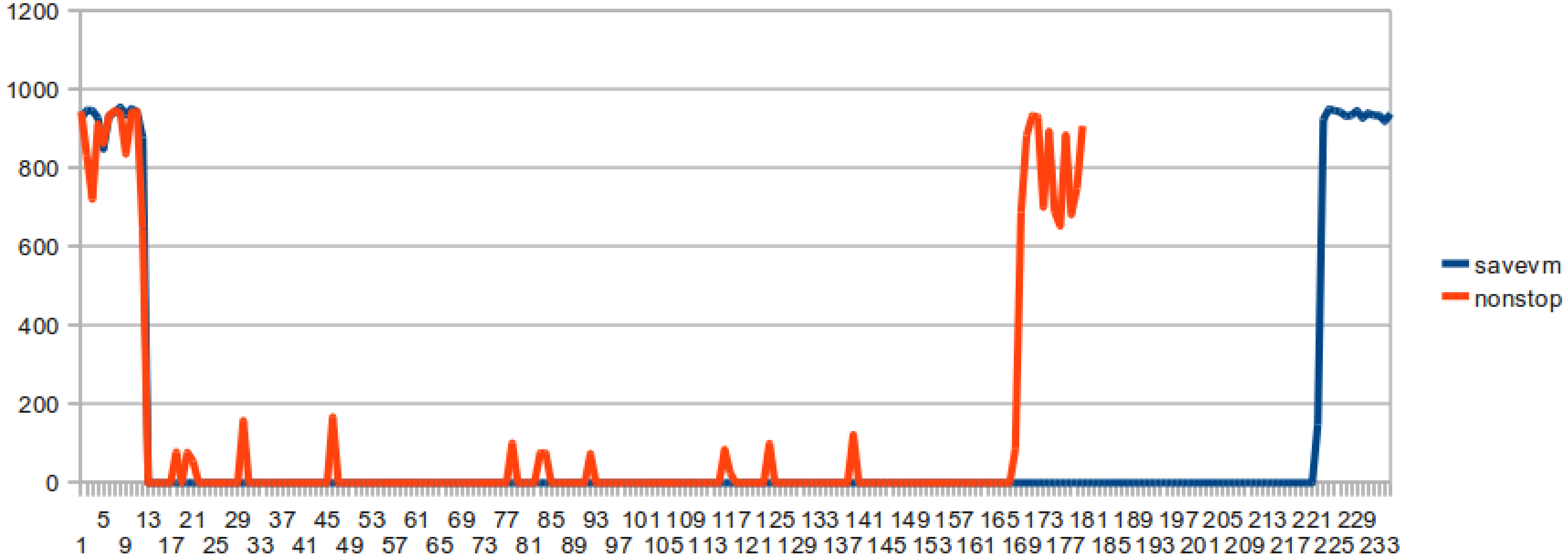
Bandwidth

* Under KVM, an IO request is responded by a VM thread and IO thread
* Since TLC use IO thread to copy dirty pages, it affects IO bandwidth
* Although TLC does not interfere with VM computation in nonstop case, it takes a lot of IO thread's time
* The IO thread become a bottleneck!!

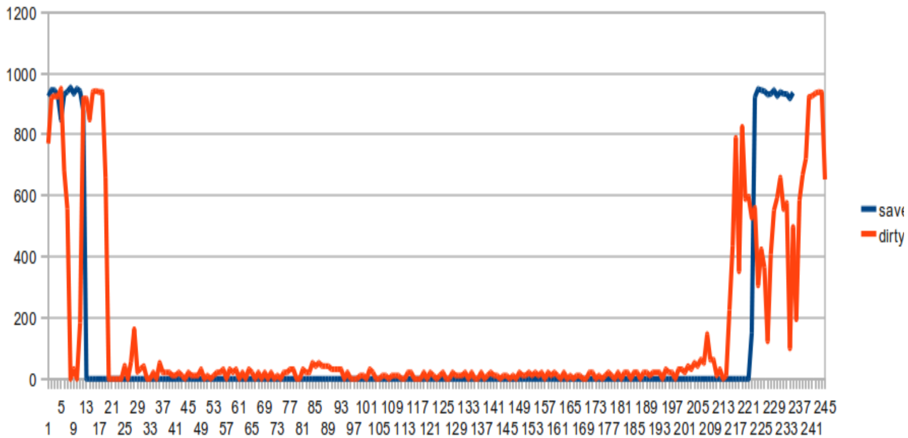
Iperf on TLC (onepass)

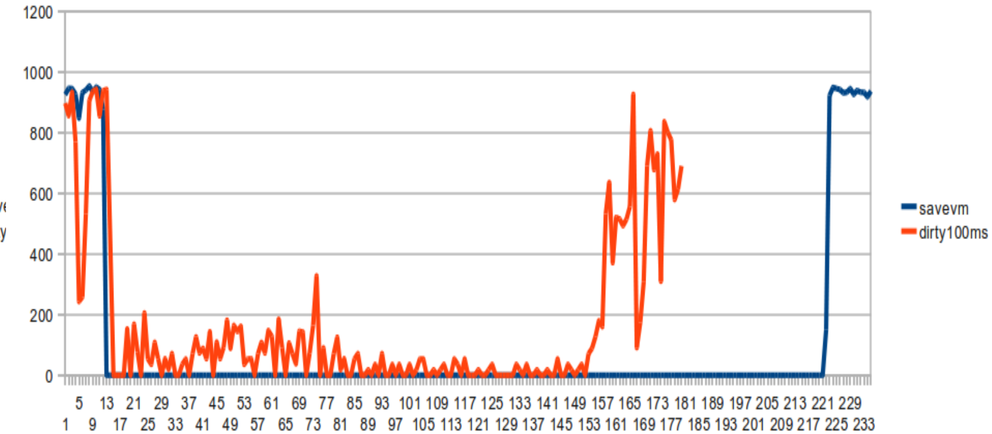


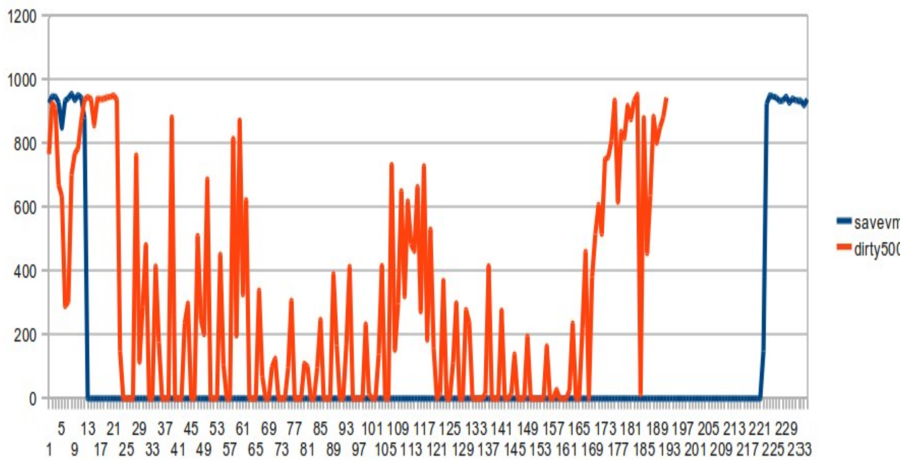
Iperf on TLC (nonstop)

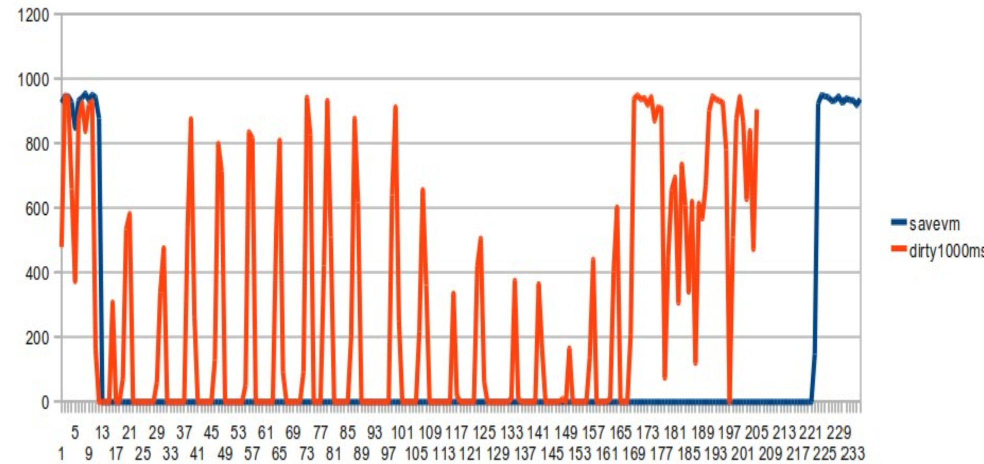


iperf on TLC (log)

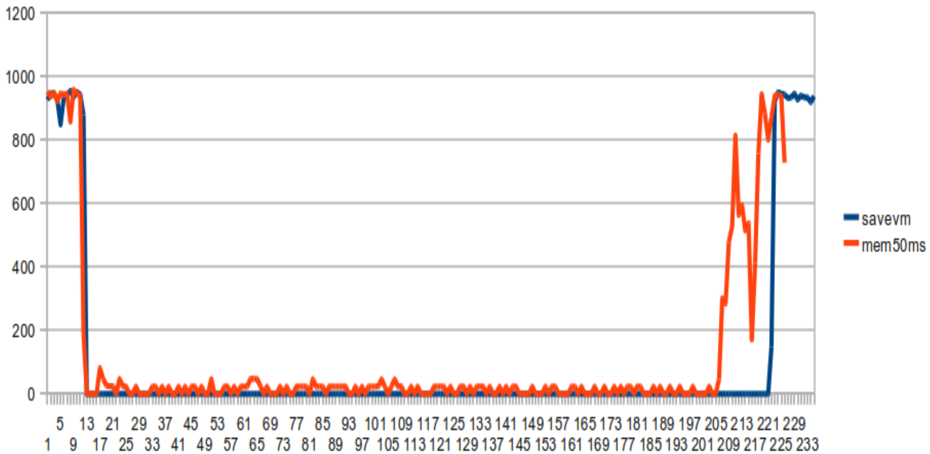


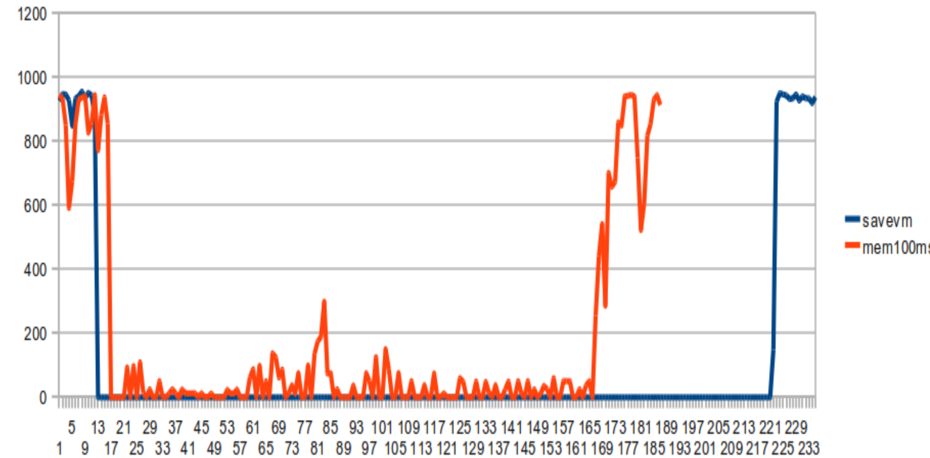


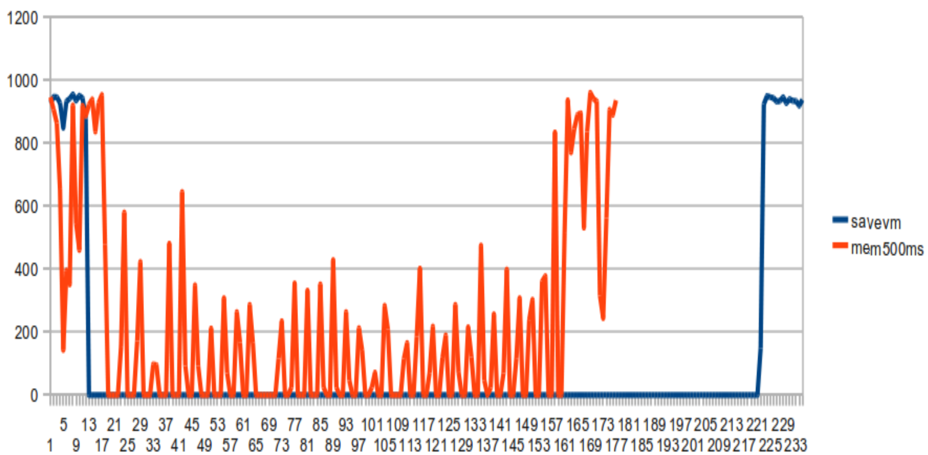


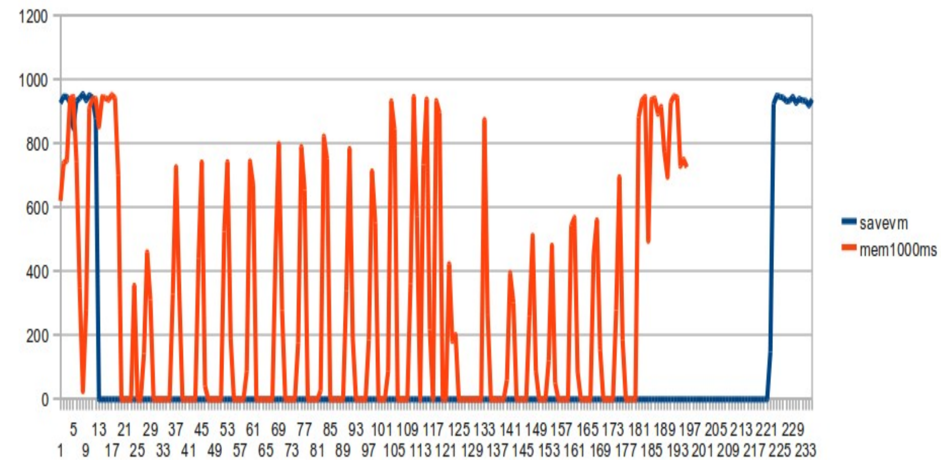


Iperf on TLC (mem)









Conclusion and Future Works

* We have developed TLC mechanism and evaluated it
* TLC is practical with computation and memory intensive workloads
* We found out that TLC service affects responsiveness because the IO thread is the bottleneck.
  - Need to modify hypervisor architecture, or
  - Configure guest to use SR-IOV
* Hash Table is Large
  - We are developing a mechanism to reduce it
* We are in progress of implementing the Thread-base Live Migration mechanism based on the same model
* We are extending TLC mechanism to support SMP VM

Thank You, Questions?