Hypervisor-Assisted Application Checkpointing in Virtualized Environments

Min Lee[*], A. S. Krishnakumar[†], P. Krishnan[†], Navjot Singh[†], Shalini Yajnik[†]
[*]*Georgia Institute of Technology, Atlanta, GA 30332*
[†]*Avaya Labs, Basking Ridge, NJ 07920*
minlee@cc.gatech.edu, {ask, pk, singh, shalini}@avaya.com

***Abstract*—**
There are two broad categories of approaches used for checkpointing: application-transparent and application-assisted. Typically, application-assisted approaches provide a more flexible and light-weight mechanism but require changes to the application. Although most applications run well under virtualization (e.g. Xen which is being adopted widely), the addition of application-assisted checkpointing - used for high availability - causes performance problems. This is due to the overhead of key system calls used by the checkpointing techniques under virtualization.

To overcome this, we introduce the notion of *hypervisor-assisted application checkpointing with no changes to the guest operating system*. We present the design and a Xen-based implementation of our family of application checkpointing techniques. Our experiments show performance improvements of 4x to 13x in the primitives used for supporting high availability compared to purely user-level approaches.

Keywords - virtualization; hypervisor; Xen; checkpointing; high-availabiliy

I. INTRODUCTION

Checkpoint/restart is one of the standard mechanisms for achieving high availability in long running computing systems [1]. The state of the application and/or the OS is either stored locally or carried over the network to a backup machine for future recovery. There has been extensive research in the area of checkpointing in the last two decades [2][3][4][5][6][7][8][9]. Libckp [2], libckpt [3], Condor checkpointing [5], are some of the initial systems that incorporated libraries for automated checkpointing.

Research literature classifies checkpointing approaches into two main categories – (a) application-transparent checkpoints [1][2][3], where the application does not need to be modified or be aware of the checkpoints happening, and (b) application-assisted checkpoints [10][11], where the application defines the data to be checkpointed and drives the checkpoints. The application-transparent approaches have the benefit of not requiring changes to the application. On the other hand, they have to checkpoint *all* the application state and incur higher performance overheads. Although the application-assisted approaches require changes to the application, they are usually more efficient since they can accurately determine the checkpoint size and frequency based on application demands. *Incremental checkpoints* [3][12] are one way to reduce checkpoint overheads. As the name suggests, instead of whole memory checkpoints, only differences from the previous state are checkpointed. Both the application-transparent and application-assisted approaches can benefit from the use of incremental checkpoints.

Virtualization technology is being widely adopted as a means for server consolidation. Most application servers deployed under virtualized environments need high availability, so that they can provide 24×7 service to their geographically diverse set of clients. The work on replica coordination techniques by Bressoud [13] was one of the first to propose high availability under virtualization. Virtualization platforms like KVM [14], VMware vSphere [15] and Xen [16], provide mechanisms like snapshots and live migration [17], for achieving high availability under failure conditions. The work by Wang et al. [18] proposes checkpointing of virtual machines using a special-purpose checkpointing VM. Remus [19] and Kemari [20] are examples of application-transparent incremental checkpointing frameworks in the Xen environment. Both techniques periodically copy the disk and memory state of the virtualized OS and the applications to a backup system. Since each checkpoint copies the entire changed state of the virtual machine, the data processing and migration overheads can be significantly high, especially for applications that need high performance and have a limited data set that they need for recovery. Our work targets such applications and focuses on application-assisted incremental checkpointing techniques.

On the face of it, application-assisted checkpoints can run unchanged on virtualized platforms. While this is functionally true, we have observed that there is a significant performance penalty arising from the inherent nature of virtualization implementation. Understanding and mitigating this issue is the main focus of our effort. Incremental checkpoints are usually implemented using a page-fault based mechanism. Pages dirtied since the last checkpoint are tracked by making them read-only and having the application/OS fault when data is written to those pages [21]. Usually in native environments, implementation of incremental checkpoints is very efficient. However, in virtualized environments, due to the overheads related to trapping multiple times to the hypervisor, the primitives used to implement the page fault mechanism become very expensive. (This is explained later in Section II.D.) Our work in this paper is targeted to address this overhead so that these primitives can be implemented efficiently under virtualization, thereby enabling application-assisted checkpointing techniques to retain their high performance.

In this paper, we introduce a new model: *hypervisor-assisted application checkpointing*. In our model, the hypervisor of the virtualization platform provides efficient primitives that assist applications to track page fault behavior. We also introduce a novel mechanism for applications to use these hypervisor-provided primitives. Rather than the usual method of only allowing a guest operating system to use hypervisor services, we enable an application running inside the guest OS to invoke these primitives in the hypervisor *directly and securely*, *without any changes to the guest operating system*. This allows for better code maintenance and easier deployment, since the underlying operating system in which the checkpointed application is deployed does not need to be changed to use our technique. Bypassing the operating system for specific features the hypervisor provides, and doing so securely is novel even from the virtualization standpoint and is motivated by application-assisted checkpointing.

We present a family of techniques that use our hypervisor-assistance paradigm and describe their implementation in the Xen virtualization platform. We have conducted detailed experiments including microbenchmark studies and performance results for basic data structure operations used in standard application transactions. Our experimental results demonstrate a significant performance improvement: specifically, a 4x-13x boost in performance in the page fault primitives that lie at the heart of application checkpointing techniques.

The rest of the paper is organized as follows. Section II introduces some basic concepts in application checkpointing and virtualization and motivates the performance problem of checkpointing under virtualization. In Section III, we introduce our model of hypervisor-assisted checkpointing, its key features, and implementation challenges. Section IV discusses our family of checkpointing approaches. Section V explores the performance of our approaches using microbenchmarks. In Section VI, we use a workload-based evaluation using data structure operations to study our techniques and conclude in Section VII.

II. Checkpointing and Virtualization

In this section, we provide a brief background of checkpointing and virtualization and outline the source of performance degradation of checkpointing under virtualization.

A. *Application-Assisted Checkpointing*

In application-assisted checkpointing techniques, the application usually defines memory areas that need to be checkpointed for recovery. We call these segments of the memory as the *critical data area* (CDA). At the end of a checkpoint cycle, the CDA is saved to disk or synchronized to the backup CDA in an atomic operation. In application-assisted checkpointing, it is the application's responsibility to determine the checkpoint cycle, i.e., the start and end of the checkpoint. Usually, an application transactionalizes certain operations or groups of operations on the critical data area by invoking checkpoint begin and end calls at transaction boundaries. This results in either all or none of the changes within a transaction being carried over to the backup.

Figure 1 shows how high availability is achieved by checkpointing data structures at transaction boundaries. In this example, each list operation in the figure is treated as a transaction by the application. At the completion of the list operation, checkpoint cycle is terminated to copy over the modified pages to the backup.

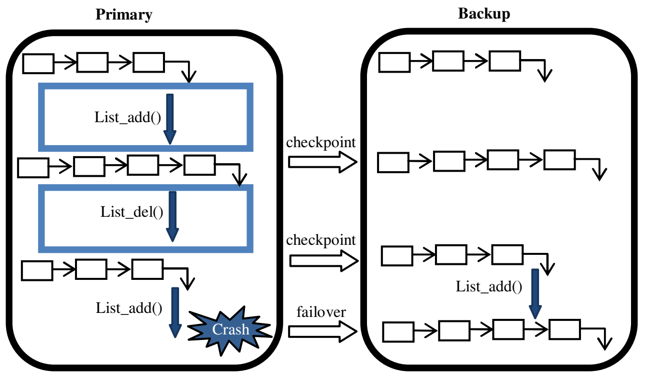


Figure 1: Checkpoint and recovery

Application-assisted checkpointing approaches usually define a simple API which provides the application the functionality to declare a critical data area (CDA) and define the start/end of a transaction or checkpoint cycle. Existing code can be modified to use these primitives to define checkpointed data and also the transaction boundaries.

B. *Incremental Checkpointing*

Incremental checkpoint minimizes checkpointing overhead by synchronizing just the pages that were modified after the last checkpoint. A *page-fault based mechanism* is typically used to determine the modified (dirty) pages [3][12][21]. At the beginning of a checkpoint cycle, all pages that are part of the CDA are write-protected by using a memory protection command (e.g. the *mprotect* system call). When the application tries to modify a write-protected page, a protection violation signal is generated. This signal can be trapped by a signal handler. The signal handler adds the address of the faulting page to the list of changed pages and removes the write protection from the page (e.g. by another call to mprotect) so that the application can proceed with the write. At the end of the checkpoint cycle, the list of changed pages contains all the pages that were modified in this checkpoint cycle. The program can then be paused momentarily to save the contents of the changed pages. We call this technique as *page-tracking (PT) based*, since it tracks changes at the page-level granularity.
Existing page-tracking techniques can use an optimization of *difference computation* to detect the changes within the page and then save only the modified words to the backup. This trades compute overhead for data reduction.

C. *Platform Virtualization with Xen*

Server consolidation to reduce cost, space and power has been a driving force behind the success of platform virtualization. Virtualization allows multiple servers to run on the same physical hardware without interfering with each other. A thin layer called hypervisor or Virtual Machine Monitor (VMM) runs on top of the hardware and provides virtual hardware interfaces to the VMs.

In the case of Xen (see Figure 2), the hypervisor (VMM) runs at the highest privilege level and controls the hardware. Virtual machine instances are also called *domains* in Xen. A privileged domain called Dom0 and other non-privileged guest domains called DomU run above the hypervisor like an application runs on an OS. Dom0 is a management domain that is privileged by Xen to directly access the hardware and it manages the initiation/termination of other domains.

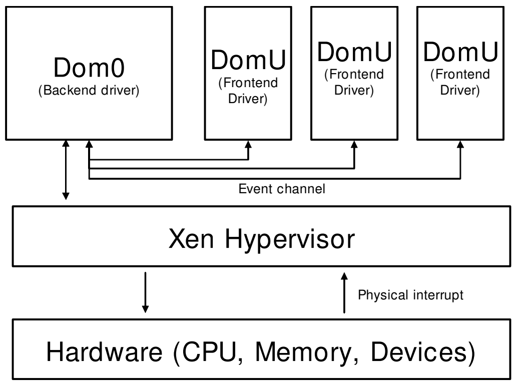


Figure 2: Virtualization with Xen

The hypervisor virtualizes physical resources such as CPUs and memory for the guest domains. Most of the non-privileged instructions can be executed by the guest domains natively without the intervention of the hypervisor. However, privileged instructions will generate a trap into the hypervisor. The hypervisor validates the request and allows it to continue. This makes certain operations such as page table manipulation especially expensive in virtualized environments. The guest domain can also use *hypercalls* to invoke functions in the hypervisor. For this, the guest OS needs to be ported to use the functionality and this porting is called *para-virtualization*. Xen provides a delegation approach for I/O via a split device driver model where each I/O device driver called the *backend* driver runs in Dom0. The DomU has a *frontend* driver that communicates with the backend driver via *event channels* and shared memory.

D. Performance Overhead of Checkpointing under Virtualization

A quick experiment of the performance of page protection in native vs. virtualized environments, both at the user level, shows that one page protection call (specifically, mprotect() calls under Xen) is *approximately 4 times slower under virtualization*. To understand why there is this enormous overhead, we need to look under the hood of how the relevant system calls operate under virtualization.

During the checkpoint interval, each time the application writes to a write-protected page, it receives a page fault that traps into the signal handler. Figure 3 shows how the page-fault is handled in native and virtual environments like Xen. Unlike the native environment, under virtualization, this call is trapped to the hypervisor. The signal handler issues a page protection call to unprotect the page. This page protection system call goes into kernel space and issues a call to update the page table. The page table update invokes a hypercall to trigger a translation look-aside buffer (TLB) flush because TLB must be flushed to be synchronized with the page table. A hypercall is needed *since the privileged page table operations can only be done in the context of the hypervisor*. The increased number of context switches between kernel-space and the hypervisor and the added overhead of scheduling each of these in the virtual environment, makes the whole cycle very expensive.

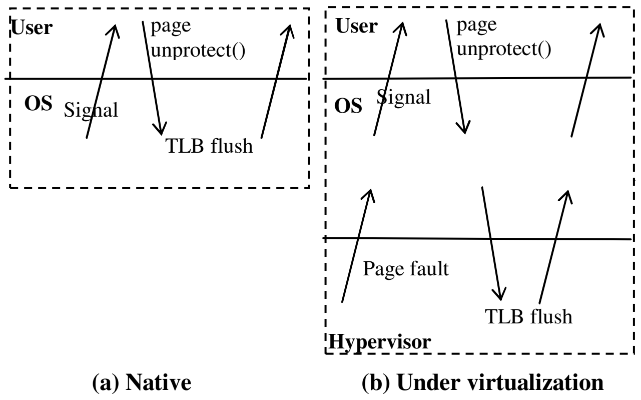


Figure 3: Page fault handling on a write operation to a protected page

Understanding this overhead and its impact on checkpointing is one contribution of our work. We now delve into our approaches for solving this issue.

III. HYPERVISOR-ASSISTED CHECKPOINTING

To tackle the significant overhead of page protection system calls in virtual environments, we introduce a new model of checkpointing: *hypervisor-assisted application checkpointing*. The model has two key aspects: (i) support in the hypervisor to speed up certain operations that are key to checkpointing, and (ii) a new model of application-hypervisor interaction motivated by our checkpointing application. An important aspect of the model is its practicality and its feasibility of implementation. To that end, along with the model we present details of how it can be implemented in a sample open-source virtualization platform (namely, Xen). Specific checkpointing techniques that use this model are discussed later in Section IV.

A. Checkpointing Support in the Hypervisor

The first aspect (namely, support in the hypervisor) involves changes to the hypervisor to provide primitives that can track pages changed in a transaction. These primitives provide the ability for the caller to inform the hypervisor about (i) the memory associated with the critical data area, and (ii) the start and end of a transaction. Implementing these APIs can be considered similar to making an ioctl() or system call to the hypervisor and the relevant data (e.g., identity of the critical data area) is passed as arguments.

At a high level, the hypervisor implements techniques to track the changes in the critical data area within a transaction and provide the caller with these changes at the end of the transaction. However, there are interesting design and implementation issues in how this is tackled by the hypervisor and we elaborate on these below.

One key issue is that the hypervisor must be able to override the application page fault handling mechanism, so that it can tackle it in the hypervisor. This is relatively simple, given the higher privilege level at which the hypervisor operates. In particular, a new page fault handler in the hypervisor checks if a fault is within the critical data area registered with it and, if so, handles the fault and returns a success so execution can proceed normally in the application. A second related design issue is *what to do in the fault handler.* One possible approach is to track the identity of the faulted page, which is a hypervisor-assisted counterpart to the *PT* approach from Section II.B and is elaborated on in Section IV.A. However, our architecture is general enough to allow the fault handling logic to be pluggable. This allows for interesting new techniques supported by our paradigm and they are described in Section IV.

Another design issue is process identification, since isolation is a key feature of virtualization that must continue to be supported. While this also appears to be straightforward, in practice it is not trivial. The currently running process is *not visible* to the hypervisor. However, there is an interesting technique based on address space changes where the address space is used to infer the identity of the process and this is discussed in more detail below in subsection D.

Hypervisors are designed to have a low footprint. Clearly, storage of too much information within the hypervisor context is undesirable. The bulk of the storage for our techniques involves tracking and maintaining the changed data through the checkpoint cycle. In our architecture, *the caller allocates space* for storing this information, and the hypervisor directly writes to that area. This obviates the need for maintaining this information within the hypervisor.

B. *Application-Hypervisor Interaction*

In our discussion above in subsection A, we deliberately avoided the issue of how the primitives provided by the hypervisor are invoked by the application. This is a crucial aspect of our technique and we elaborate on that below.

A *hypercall* is a software trap from a guest OS to the hypervisor, just as a syscall is a software trap from an application to the kernel. Guest domains use hypercalls to request privileged operations like updating the page-tables. Traditionally hypercalls are only possible from inside the guest operating system. Applications are not allowed to invoke hypercalls directly. The traditional approach would, therefore, create corresponding system calls in the guest operating system that will be invoked by the application, which would then translate to our checkpointing hypercalls. Although potential performance benefits may still be realized by this implementation, there is a deployment issue. Changing the guest operating system for each deployment supported by the application is non-trivial.

To tackle this issue, we introduce the concept of *secure direct hypervisor calls from the application.* This is useful when a guest domain needs to be deployed using an unmodified guest OS. In this model, the application directly talks to the hypervisor bypassing the guest OS. This model of communication is also novel from a virtualization perspective.

There are a few ways in which the model can be implemented. Regular system calls and hypercalls from the guest operating system are traditionally implemented in x86 architectures via an interrupt vector with values 0x80 and 0x82 respectively. We have implemented the user-to-hypervisor call through an additional interrupt vector 0x84 as shown below.

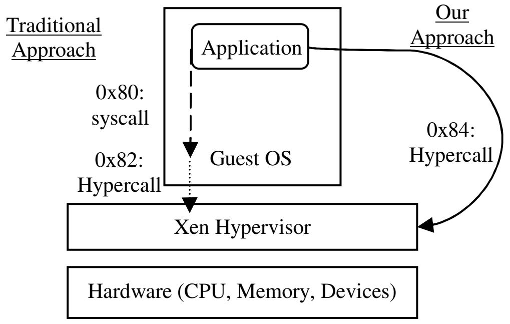


Figure 4: User-to-Hypervisor Call

For security purposes, only a set of pre-defined hypercalls are allowed to use the 0x84 interrupt vector. Additionally, these hypercalls are only allowed to work in the process space of the calling process, thereby creating a level of isolation essential for security. (Isolation is obtained using the techniques in subsection D.)

There are alternative approaches. For example, communication between the application and the hypervisor could be done through a shared memory that is communicated by the application to the hypervisor through a privileged domain like Dom0 in Xen. For brevity, we do not elaborate on these alternative approaches.

C. *Access Control*

It is possible that administrators would like to limit the application instances that can invoke the hypervisor-assisted checkpointing primitives. Authorization of valid applications can be done by using a policy module in Dom0. To achieve this, the application inside the guest domain can be provisioned with a key, and this key can be used to authenticate it to the hypervisor via a privileged domain like Dom0. The application can initiate the process via a network connection to Dom0. The privileged domain Dom0 can provide the mechanisms for registering the application and issuing any required shared tokens. Once the application is registered with Dom0, it is allowed to invoke the hypercalls directly.

D. *Implementation: Process tracking*

As discussed earlier, a user space process in the guest OS is allowed to make hypercalls to invoke functionality directly in the hypervisor. For security and functionality, Xen needs to uniquely identify the user process when it makes a hypercall. This requires guest process tracking at the hypervisor-level.

In Linux, each process has a unique address space and our technique uses this for identifying the process within the hypervisor [22][23]. Our technique is based on the fact that the address space change is visible to the hypervisor although a guest process or a task is not. For example, on x86 architectures, a new value being loaded onto a cr3 register (page directory) indicates loading of a new address space and this action is done by the hypervisor. When the guest installs a new value into the cr3 register, Xen validates this entry. This indicates the creation of a new guest process to the hypervisor. Similarly, when the guest process terminates, its address space is torn down and the pages are unmarked and returned to the guest operating system. This is also tracked by the hypervisor. In practice, this simple method works well [22][23] for tracking the identity of a user-space process.

IV. OUR APPROACHES

In this section, we introduce our incremental checkpointing approaches. In addition to hypervisor-assisted page-tracking based approach (PTxen), we also introduce a new concept of emulation-based approaches. Emulation-based approaches for checkpointing have not been studied in earlier literature and both hypervisor-assisted (Emulxen) and user-level (Emul) emulation techniques are introduced in this paper. Additionally, motivated by live migration techniques in Xen [17], we present a page-table scanning based approach that we call Scanxen.

Table 1 below gives a high-level categorization of our approaches and existing approaches (prior work in italics). Hypervisor-assisted approaches implement most of their functionality in the hypervisor while user-space approaches do so in pure user space. Scan-based approaches need full support from the hypervisor and do not have an equivalent implementation in user-space.

	Page-tracking based	**Emulation-based**	**Scan-based**
Pure user space	*Page-tracking (PT)*	Emul	
Hypervisor-assisted	PTxen	Emulxen	Scanxen

Table 1: Categorization of checkpoint approaches

A. *Page-tracking based hypervisor-assisted: PTxen*

PTxen is a page-tracking based approach similar to the PT technique presented in Section II.B, but implemented mostly inside the hypervisor. At the beginning, when the application declares its critical data area, the hypervisor installs the address ranges for the critical data area in an internal data structure. At the start of each checkpoint cycle, the hypervisor write-protects all the pages in the critical data area for the application. The hypervisor also overrides the standard page-fault handler to trap any writes to the pages. When the application writes to a page, the page fault handler within the hypervisor is invoked. This in turn puts the page in the modified page list and unprotects the page.

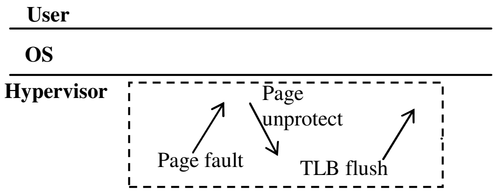


Figure 5: Page fault handling with PTxen

Figure 5 shows the operation of PTxen. As seen in the figure, the page-fault is trapped by the hypervisor and operated on in that layer, instead of propagating it to the user-space application. The simple call flow eliminates a number of overheads associated with a single write operation. As compared to the call flow in Figure 3(b), the numerous context switches between user-space, guest OS and the hypervisor (as experienced in the standard PT case) are eliminated. These are replaced by the majority of work being done in the hypervisor, thereby reducing the context-switch and scheduling overhead and multiple calls to page protection by the application. Additionally, in comparison to Figure 3(a), we see that the hypervisor-assisted model does page protection *at a lower layer* (hypervisor) than the native case (application layer) allowing for the possibility that its performance can be *even better* than native performance.

PTxen can work in parallel with other techniques like live migration [17]. Since both pieces of code are implemented in the hypervisor and override the page fault handler, they can be combined to coexist in the hypervisor.

B. *Emulation-based: Emul*

The page-tracking approach discussed above dealt with changes at the granularity of pages. An emulation-based technique deals with changes being maintained at the word-level granularity.

At a high level, emulation-based approaches also depend on a page-fault mechanism for tracking changes. Once a page-fault is detected they operate at the granularity of a word. They write-protect the critical data area at the beginning. A separate unprotected mapping (e.g. via *mmap*) is maintained for the CDA. When the application writes to the protected area, the system generates a protection violation which is then communicated to the application. Within the signal handler, the application detects the word that is written to, makes a copy of the changed word and then writes to the critical data area using the alternate mapping *without unprotecting the page*. In x86 architectures, the write is emulated using the x86 'MOV' instruction so the data is written one-word at a time. At the end of the checkpoint cycle, the application has a list of all the changed words and can use this list to build a checkpoint. Since the list is maintained at the word level, only the data that has really been modified needs to be migrated to the backup, thereby saving bandwidth and compute power.

C. *Emulation-based hypervisor-assisted: Emulxen*

Emulxen is the hypervisor-assisted version of the emulation approach discussed above in Subsection B. When the application declares its critical data area, the hypervisor write-protects all the pages in the critical data area. Similar to the PTxen case, the hypervisor overwrites the page fault handler to trap all page faults locally. When the application writes to the critical data area, the system generates a page-fault which is trapped by the page-fault handler in the hypervisor. The page fault handler notes the address and the value of the dirty words and records them in a buffer provided by the application. It then emulates the write as with Emul. At the end of the checkpoint cycle, the hypervisor has the full list of changed words and the values of the changed words in the buffer in application space.

D. *Scan-based hypervisor-assisted: Scanxen*

The scan-based approach is motivated by live migration in Xen [17]. Instead of protecting and unprotecting pages explicitly, the technique is based on scanning page table's dirty bits to obtain a list of modified pages. When the application declares its critical data areas, the hypervisor keeps the critical data areas in its list of pages to track. Whenever the application writes to a page, the hardware tracks the write by setting the dirty bit in the page table. However, in normal systems, the dirty bit would be reset as soon as the page is swapped to disk. Xen supports the concept of *shadow-page tables* where the guest OS uses a copy of the page tables that is independent of the hardware page tables. Xen propagates the changes made to the shadow page tables to the hardware page tables and vice versa. Scanxen uses the dirty bits in the shadow-page tables to track the modified pages. (In contrast, PT-based and Emulation-based do not rely on shadow-page tables, but maintain their own dirty pages.) At the end of the checkpoint cycle, Scanxen parses the guest OS shadow page table to determine the set of dirty bits in the critical data area for a given application. It builds a list of changed pages from this and passes it to the application. For performance reasons, in our implementation, we did not use the "log dirty bit" facility from Xen live migration for maintaining the dirty bits, but constructed them directly from the shadow page tables.

Note that for each checkpoint cycle, Scanxen has to walk through the guest OS page table and access all the pages in the critical data area. The cost of Scanxen depends on the size of the critical data area, and not on the number of dirty pages/words in a transaction. This can be expensive if the critical data area buffer is large.

V. MICROBENCHMARK

In order to evaluate the performance of each approach we built a microbenchmark. Memory-write operations have a direct impact on the checkpoint performance; hence the microbenchmark first allocates a critical data area and then performs a number of memory write operations. It transactionalizes each write or a group of writes in the CDA by containing them between checkpoint begin and end calls. Four key parameters were used to parameterize the benchmarks:

- Size of the critical data area (CDA)
- Writes-per-page (WPP): Average number of write operations on a page within a transaction.
- Pages-per-transaction (PPT): Average number of unique pages written to in each transaction (checkpoint cycle).
- Transaction count (Tcount): Total number of transactions (checkpoint cycles) in the experiment.

Total size of a transaction (Tsize) is defined as the total number of writes in a transaction which is the product of writes-per-page (WPP) and pages-per-transaction (PPT):

$$\text{Tsize} = \text{WPP*PPT}.$$

The results in this section show the time taken for Tcount = 100000 transactions.

In this section, we assess the impact of the above parameters on the performance of our approaches. The evaluation is useful in understanding which approaches are better fitted to certain types of transactions. The experiments have been performed with Xen 4.1-unstable. The Dom0 kernel was 64-bit Linux 2.6.32-15 and the guest kernel was paravirtualized 32-bit Linux 2.6.18-164. Both Dom0 and guest kernels were patched with pvops kernel patches [24].

A. *PT-based approaches*

Figure 6 shows the performance of the two page-tracking based approaches (PT and PTxen) with varying PPT and WPP. Note that the three runs with different WPPs (4, 8, and 16) all have the same result for a given approach. This is to be expected since varying the WPP for each approach has no impact on the performance of the approach. Varying the PPT has a direct impact on the performance of the approach. This is because page-tracking based approaches incur an overhead each time a page is dirtied for the first time with a transaction. Once the page is unprotected and written to, there is no additional cost for subsequent writes into the page. Hence there is a linear increase in overhead with PPT. An important result from this experiment is that PTxen shows a tenfold improvement in performance, thereby validating the hypervisor-assisted approach.

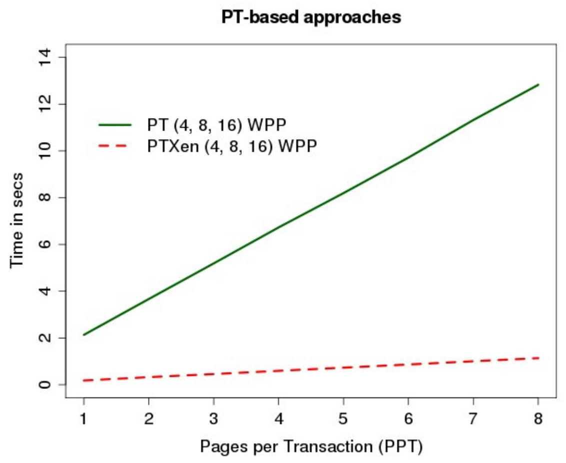


Figure 6: PT vs. PTxen with varying PPT

B. Emulation-based approaches

Figure 7 shows that the emulation-based approaches get impacted mainly by transaction size (WPP*PPT) rather than individual values of WPP or PPT. This is because emulation-based approaches emulate every write into a page, be it the first write or a subsequent write. So the performance doesn't depend on pages modified per transaction but more on the total number of writes within a transaction. This has advantages if the transaction has high PPT and low WPP. If WPP is low, emulation-based approaches can eliminate unnecessary page-table manipulations and have the potential to outperform page-tracking based approaches.

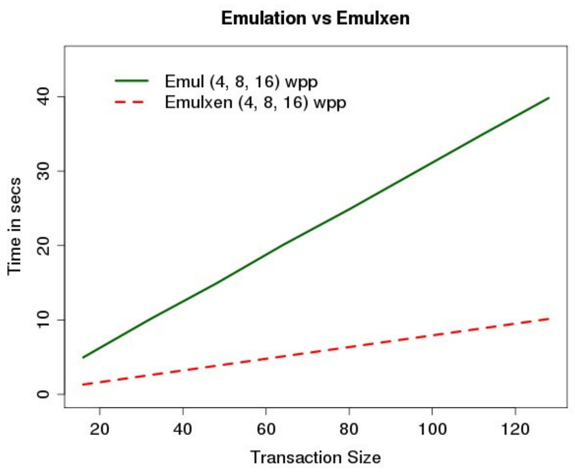


Figure 7: Emul vs. Emulxen with varying Tsize

Applications with simple operations that have a small number of writes within a transaction, such as list deletions, are a good candidate for emulation-based approaches.

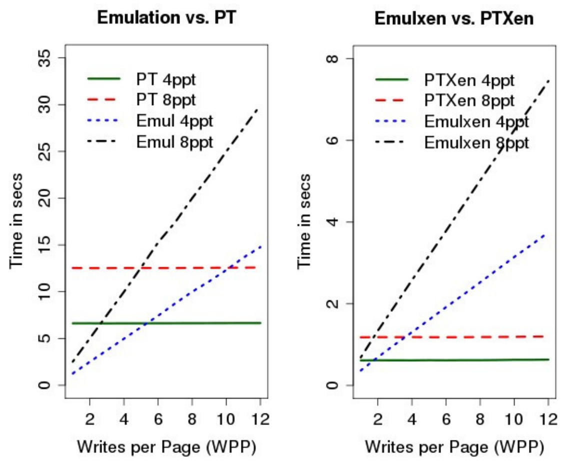
As shown in the figure, a comparison between Emul and Emulxen shows a fourfold improvement from Emul to Emulxen, further validating the efficacy of the hypervisor-assisted model.

C. Emulation vs Page-tracking

As discussed earlier, emulation is good for small transactions or transactions with small number of writes per page. In this subsection we investigate the break-even point between emulation and page-tracking based approaches.

Figure 8 shows the comparison between emulation-based approaches and page-tracking based approaches. The main aim of the experiment is to find the optimal parameter values for the two categories of approaches. Figure 8(a) shows that in user-space, below 5 writes-per-page (WPP), emulation performs better than PT. Figure 8(b) shows that, for hypervisor-assisted approaches, Emulxen performs better than PTxen for WPP below 1.3. Beyond these two numbers, the page-tracking based approaches have better performance.

The results show that in the user-space, five write emulations and page faults are equivalent to a single page protection and page fault. Compared to user-space case, hypervisor-assisted case shows a much lower break-even point (WPP = 1.3). This illustrates the significant overhead of page fault handling in user space.

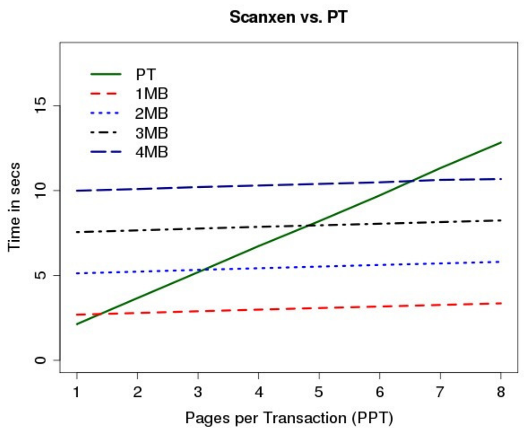


(a) User level **(b) hypervisor-assisted**

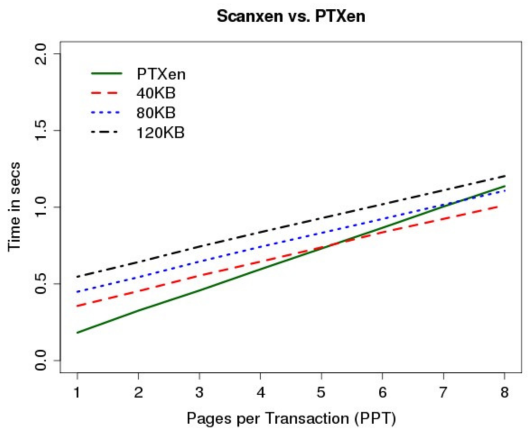
Figure 8: Emulation vs. Page-tracking

D. Scanxen

As discussed in the earlier sections, Scanxen is *mostly* dependent on the size of the critical data area. The main overhead of Scanxen comes from scanning the page tables to get the dirty bits.



(a) vs. User-level



(b) vs. Hypervisor-assisted

Figure 9: Scanxen performance

Figure 9 shows the performance of Scanxen with respect to PT (Figure 9(a)) and PTxen (Figure 9(b)). Although most of the Scanxen overhead comes from scanning the page-tables, there is some impact of pages-per-transaction (PPT) as can be seen from the positive slope of Scanxen lines in the two figures. At the end of each transaction, Scanxen constructs the list of dirty pages. The work involved in building the list is proportional to the number of dirty pages (PPT). The total cost y can be expressed with a simple linear equation:

$$y = (\text{PPT dependent cost}) + \text{static cost based on CDA}.$$

Based on Figure 9, we find that:

$$y = (0.0625)*(\text{PPT}) + 2.5*(\text{size of CDA in MB}),$$

where the first term represents the amount of work to be done for accumulating the list of dirty pages.

For large critical data areas (e.g., several Mbytes) the static cost is dominant, so the first term in the equation is negligible. On the other hand when the critical data area is small (e.g., several tens of Kbytes) the first term has a bigger impact on the overall cost.

The two figures show the break-even points for Scanxen when compared to the page-tracking based approaches. As compared to PT, Scanxen performs better with higher values of PPT and for smaller CDAs. In hypervisor-assisted page-tracking case (PTxen), due to the improved performance of PTxen, Scanxen cannot outperform it in most cases, except when the CDA is smaller (10s or 100s of Kbytes).

Although Scanxen can be better in performance for applications with small size CDA and large PPT transactions, the range of values for which it is better is so small that in the practical case most applications do not fit the criteria. For most real-world applications, PT and PTxen can easily outperform Scanxen. In this work we will not present additional results on Scanxen.

E. *Summary*

Figure 10 summarizes the performance of the various approaches for a sample case of WPP=4. Overall, we note that the hypervisor-assisted approaches are 4-10x better in performance than user-level approaches.

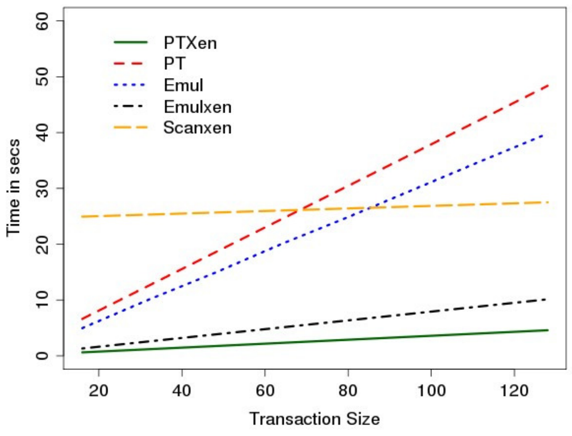


Figure 10: Comparison of approaches

One interesting observation from the figure is that while Emul is better than PT in the user space under virtualization (at least for WPP=4), PTxen is better than Emulxen. This suggests that the gains in moving page protection to hypervisor space are especially significant, making page tracking-based approaches with hypervisor assistance outperform other techniques.

VI. Workload Evaluation

To evaluate a more realistic workload, we implemented data structures typically used in most applications [25]. We studied two cases – (a) where each transaction had a single operation (e.g. an insert or a delete), i.e. Operations-per-Transaction (OPT) is 1, and (b) where multiple operations were merged into one transaction, specifically OPT = 5.

Table 2 gives a list of the data structures implemented. For each data structure it shows the average number of data writes and the average number of unique pages written to in a transaction by insert and delete operations. In the case of OPT=1 the numbers are for a single operation and when OPT=5 it is for five operations. In the workload experiment 10000 unique data structure operations were performed, resulting in 10000 transactions for OPT = 1 and in 2000 transactions for OPT=5. As an example, in the case of AVL tree data structure with OPT=1, each data structure insert operation created on average 30.5 writes and on average 5.1 unique pages were modified. As expected, the number of write operations increases approximately five times between OPT=1 and OPT=5. However, the number of unique pages touched by OPT=5 does not grow linearly with respect to OPT=1. In fact, in most cases, the number of unique pages touched is approximately the same. This is because the multiple operations within the transaction may touch the same pages several times.

Table 2: Data Structures and Operations

Data Structures		**OPT=1**		**OPT=5**	
	ops	**Avg. writes**	**Avg. pages**	**Avg. writes**	**Avg. pages**
aa (AA-trees)	insert	21.9	4.9	109.9	5.0
	delete	20.4	6.0	102.0	8.8
avl (AVL trees)	insert	30.5	5.1	152.8	5.1
bin (Binomial queue)	insert	27.9	2.0	139.9	2.3
dsl	insert	10.4	3.1	52.0	3.6
hashquad	insert	11.3	1.0	56.9	1.6
hashsepchain	insert	4	1.9	20	1.9
leftheap (Leftist heap)	insert	23.5	3.0	117.8	3.0
	delete	34.0	9.2	170.0	18.5
heap (binary heaps)	insert	2.8	2.4	14.3	2.8
	delete	12.5	2.7	62.7	4.1
list (Linked list)	insert	4	1.0	20	1.0
	delete	1	1	5	1
queue (Queues)	insert	3	1.8	15	1.9
	delete	2	1	10	1
rb (Red black tree)	insert	13.7	4.6	68.5	4.9
splay (Splay trees)	insert	20.0	4.7	100.4	5.0
	delete	7.7	3.0	38.8	6.7
tree (Binary search tree)	insert	720.7	5.4	3603.9	5.5
	delete	1.7	1.7	8.56	4.1

Dsl=Deterministic skip list
Hashquad=Quadratic probing hash
Hashsepchain=Separate chaining hash

A. *Performance with OPT=1*

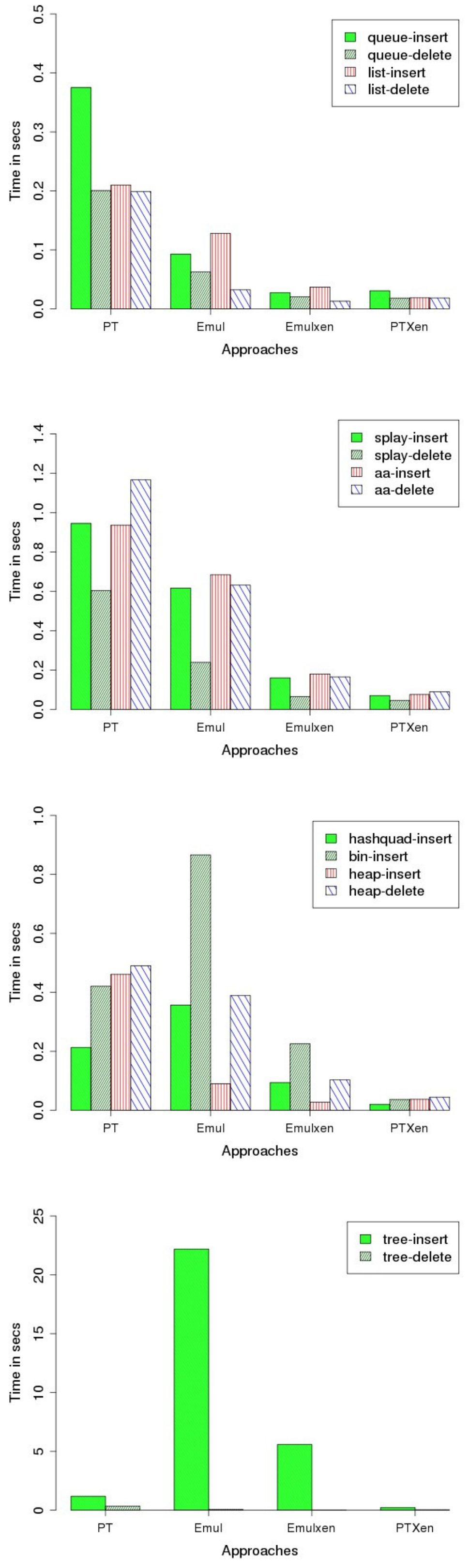


Figure 11: Workload performance

Figure 11 shows the performance of some representative sets of data structure operations from the table. Note that results for all data structures are not shown due to space constraints and the fact that they were very similar to the ones in the figure. For most operations including the queue, list, heap, splay and aa shown in the figures, performance improves from PT to Emul to Emulxen to PTxen with PTxen being the best in most cases, although there are some exceptions. Results for hashquad, bin and tree show that Emul is more expensive than PT. This is because these operations have a high write rate (high WPP) and a low number of unique pages written to (low PPT) in a transaction. As discussed earlier, page-tracking based approaches outperform emulation-based approaches for applications with such characteristics. A tree-insert operation has a very high value of WPP = 720.7/5.4 = 133.4, giving the emulation based approaches a very high overhead.

Comparing insert and delete operation for lists for Emul and Emulxen, we observe that insert operations are more expensive than delete operations. This is because insert operations incur more memory writes than delete operations. In general, the overhead of emulation based approaches is proportional to the number of memory writes.

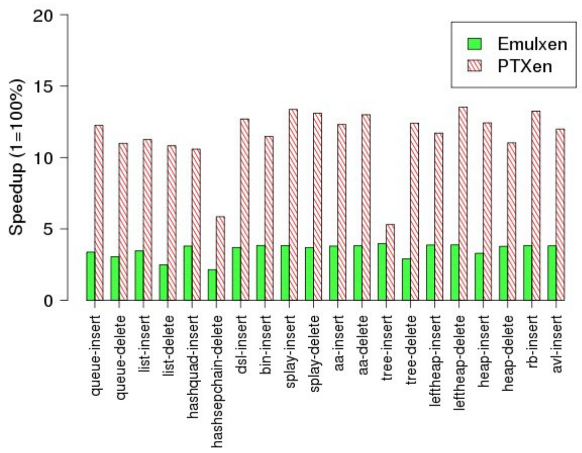


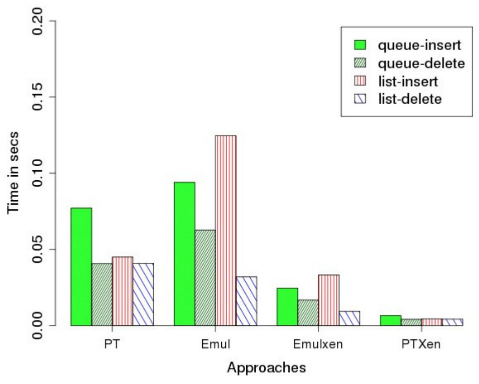
Figure 12: Speedup from hypervisor-assistance (OPT=1)

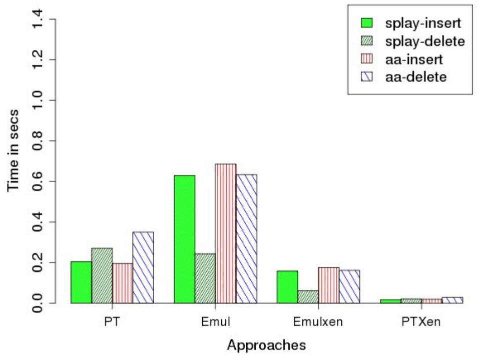
Figure 12 shows the speedup of hypervisor-assisted approaches compared to their user-level counterparts. Emulxen shows up to 4x speedup (an average speedup of 3.5 across data structures) and PTxen shows a speedup of up to 13x (an average speedup of 11.4 across all data structures).

B. *Transaction aggregation (OPT=5)*

As shown in Table 2, aggregating operations to create bigger transactions increases the number of writes linearly. However, the number of unique pages modified by the operations remains unchanged in most cases. This is because most of the operations modify data on the same set of pages. This implies that aggregating operations to create bigger transactions should benefit page-tracking based approaches, because of their heavy dependence on PPT. Emulation-based approaches which are independent of PPT, but dependent on

WPP, will have the same performance as before. In this subsection we evaluate bigger transactions where OPT=5.





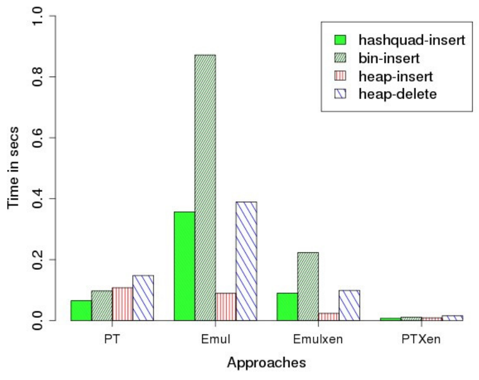


Figure 13: Data structure performance with OPT=5

As shown in Figure 13, in most cases, the user level implementation of the page tracking based approach (PT) surpasses the performance of emulation-based user-level approach (Emul). This is in contrast to Figure 11, where Emul out-performed PT in a large number of cases, showing the effect of decreased PPT on the performance of page-tracking based approaches. As compared to Figure 11, graphs in Figure 13 for page-tracking based approaches show a speedup of five times, whereas emulation based approaches do not show any difference in performance.

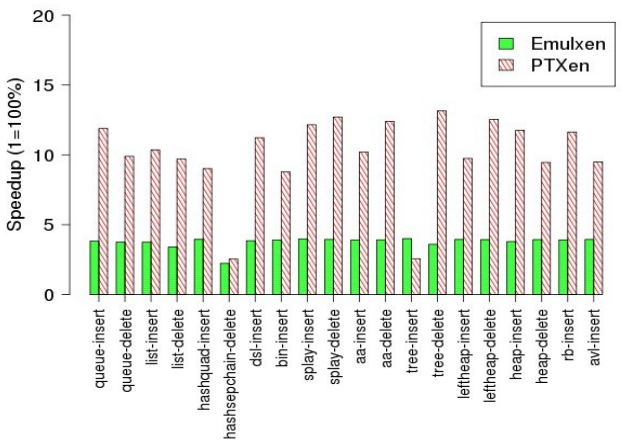


Figure 14: Speedup with hypervisor assistance (OPT=5)

Figure 14 shows the speedup of hypervisor assisted approaches over their user-level counterparts for OPT=5. As expected, the results are similar to the case of OPT=1 shown in Figure 12. This is because both PT and PTxen get an improvement of 5 times with transaction aggregation, making their relative performance same as in the case of OPT=1. Emulxen shows a speedup of up to 4 times and PTxen shows a speedup of up to 13 times over their user-level counterparts.

C. *Data Processing Overhead*

At the end of the checkpoint cycle, the modified blocks of critical data area need to be either stored to disk or transferred to the backup machine over a network. Page tracking and emulation-based techniques have to process different amounts of data. In our experiments reported here, we consider the case where OPT=1. Transaction aggregation, as noted earlier, would give better performance (i.e., lower data processing overhead) for page tracking-based approaches.

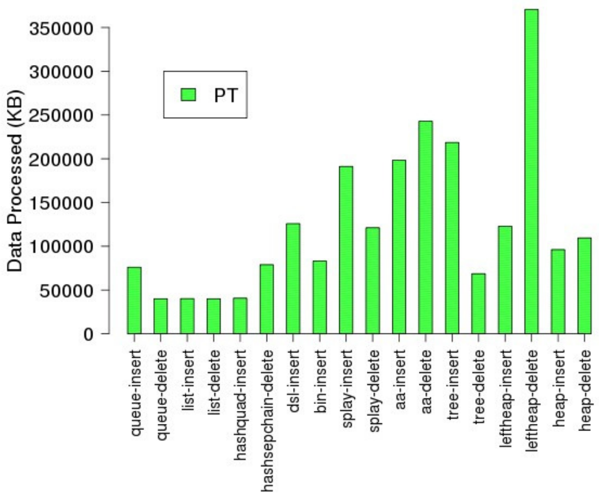


Figure 15: Amount of data processed: Page-tracking

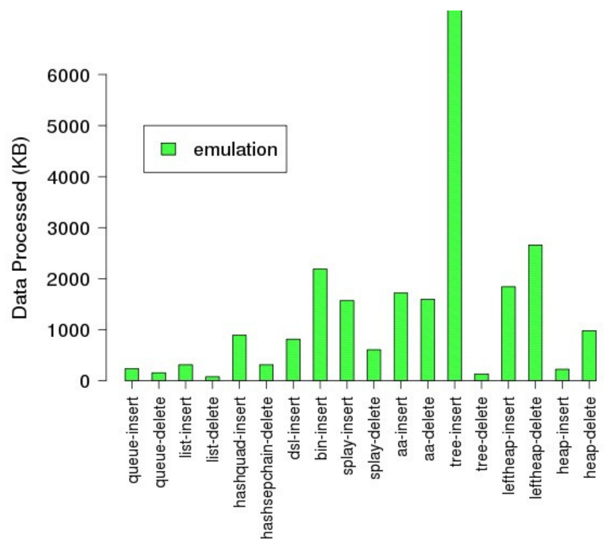


Figure 16: Amount of data processed: Emulation

Figure 15 and Figure 16 show the amount of data that each technique handles. The amount of data processed by page-tracking based approaches is of the order of 100s of MB. In contrast, for emulation-based approaches in most cases the data to be processed is less than 2MB (Note that in case of tree-insert, the value of 56MB is too large to show in scale.)

Typical implementations do not have the main process handle the data processing. The job of writing to disk or copying over to the backup is typically done by another thread or helper process. In multi-core machines, the helper process can run in parallel on an additional CPU core. In this case, the main process just copies the data (dirty pages or changed bytes) into a shared buffer. The helper process runs in parallel without stalling the main process. In this case, the only additional overhead incurred by the main application is in copying the modified data to the helper process. The helper process can either save the modified data as-is, or can do further processing on the data (e.g. difference computation for page-tracking based approaches, potential data compression, encryption for security). In this work, we focus on and evaluate the overhead incurred by the main application (namely, the cost of a memory copy).

In the case of page-tracking based approaches, the main application incurs the overhead of copying the modified pages to the helper process. Since it does not keep track of changes made within the page, it needs to copy the dirty page as a whole to the helper. In contrast, the emulation-based approaches keep track of modifications at the word granularity, so the application in this case needs to copy only the modified words to the helper.

Figure 17 shows *total time* spent (time for the 10,000 operations and the data copy to the helper process) by each approach, again for the case where OPT=1. We note that in most cases, emulation-based approaches take less than 5ms (average < 1ms) whereas page-tracking based approaches have a higher overhead in time ranging from 10ms to 80ms. However, when we look at the total time metric in Figure 17, we see that the results of Section VI.A are still valid: in effect, PTxen has overall the best performance. More importantly, the hypervisor-assisted approaches are significantly better than the user-space approaches.

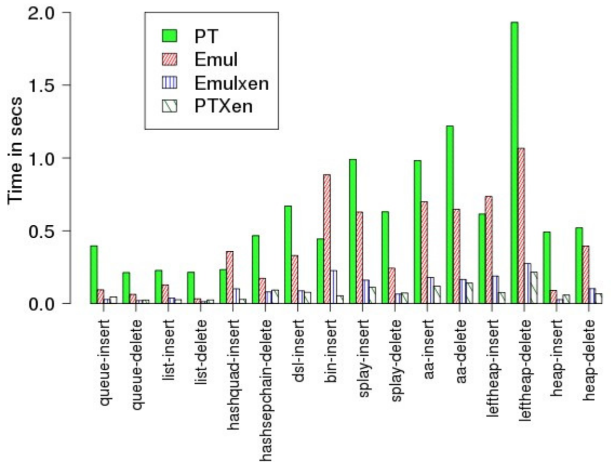


Figure 17: Total time for each approach

It is instructive to compare the overall improvement in performance due to hypervisor-assistance, taking memory copy overhead also into account. This is shown in Figure 18. We observe that PTxen improves in performance over PT by approximately a factor of 8, while Emulxen improves over Emul by approximately a factor of 4. This is a little lower than the improvements seen in Figure 12 due to the constant overhead of memory copy. Transaction aggregation (e.g., OPT=5) will clearly increase the benefits of PTxen over PT, since page reuse within a larger transaction will reduce the amount of data copied.

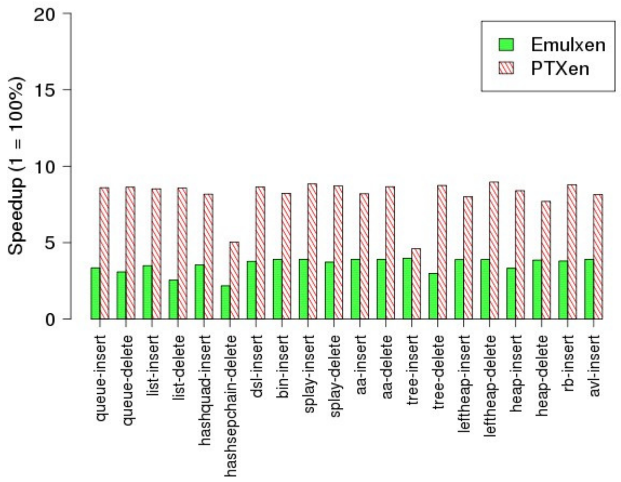


Figure 18: Net speedup of hypervisor-assisted over user space approaches

VII. Conclusion

In this paper, we discussed application-assisted checkpointing in virtualized environments. We identified the root cause of the performance bottleneck of application checkpointing under virtualization. To overcome this bottleneck, we introduced the notion of hypervisor-assisted application checkpointing. Our approach implements key primitives for application checkpointing within the hypervisor. Additionally, our approach introduces the notion of direct and secure application-to-hypervisor interaction allowing deployment with no changes to the guest operating system. Our techniques can also be applied to non-virtualized environments by incorporating them into the OS instead of the hypervisor.

We have designed and implemented a family of application checkpointing techniques. Our techniques are very lightweight and can be implemented with minimal code; e.g. our prototype for the Xen hypervisor added a few hundred lines of code totaling about 0.2% of the hypervisor code. We have introduced emulation-based techniques that are useful for small transactions. Page tracking approaches with hypervisor assistance show the best result. Compared to user-space implementations, our hypervisor-assisted application checkpointing shows impressive performance gains of 4x~10x based on microbenchmark results and 4x~13x based on results from our workload evaluation.

References

[1] E.N. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems", *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375-408, 2002.

[2] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, C. Kintala, "Checkpointing and Its Applications,", Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS), 1995, Pasadena, CA.

[3] Plank, J.S. and Kai Li, "Libckpt: Transparent Checkpointing under Unix", Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995.

[4] Jason Ansel, Kapil Arya, and Gene Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop" 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), Rome, Italy, May, 2009.

[5] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system". Technical Report CS-TR-199701346, University of Wisconsin, Madison, 1997.

[6] Hua Zhong and Jason Nieh, "CRAK: Linux Checkpoint / Restart As a Kernel Module". Technical Report CUCS-014-01. Department of Computer Science. Columbia University, November 2002.

[7] Oren Laadan and Jason Nieh, "Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems", *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 17-22, 2007, pp. 323-336.

[8] J. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner, "Cruz: Application-Transparent Distributed Checkpoint- Restart on Standard Operting Systems". In Proceedings of the International Conference on Dependable Systems and Networks (DSN'05), Yokohama, Japan, June 2005.

[9] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems". ACM Transactions on Computer Systems, 3(1):63–75, Feb. 1985.

[10] G. Deconinck , J. Vounckx , R. Lauwereins , J. A. Peperstraete, "A User-Triggered Checkpointing Library for Computation-Intensive Applications", In Proceedings of 7th IASTED-ISMM International Conference On Parallel and Distributed Computing and Systems (IASTED, Anaheim-Calgary-Zurich) (ISCC97).

[11] L.M. Silva and J.G. Silva, "System-Level Versus User-Defined Checkpointing", SRDS '98 Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems.

[12] Junyoung Heo , Sangho Yi , Yookun Cho , Jiman Hong , Sung Y. Shin, "Space-efficient page-level incremental checkpointing", Proceedings of the 2005 ACM symposium on Applied computing, March 13-17, 2005, Santa Fe, New Mexico.

[13] Thomas C. Bressoud, "Hypervisor-based Fault-tolerance", Proceedings of the 15th ACM symposium on operating systems principles, Vol. 29, No. 5. (December 1995), pp. 1-11.

[14] Kernel-based Virtual Machine (KVM) for Linux, http://www.linux-kvm.org, Last accessed on April 12, 2011.

[15] VMware vSphere – VMware virtualization platform, http://www.vmware.com/products/vsphere/overview.html, Last accessed on April 12, 2011.

[16] Xen 4.1, "http://www.xen.org/files/Xen_4_1_Datasheet.pdf" Last accessed on April 12, 2011.

[17] C. Clark et al, "Live Migration of Virtual Machines", Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2005, pp. 273-286.

[18] L. Wang, Z. Kalbarczyk, R.K. Iyer, A. Iyengar, "Checkpointing virtual machines against transient errors," Proceedings of the IEEE 16th International On-Line Testing Symposium (IOLTS), pp.97-102, July 2010.

[19] Brendan Cully et al., "Remus: high availability via asynchronous virtual machine replication", In NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (2008), pp. 161-174.

[20] Y. Tamura, "Kemari: Virtual Machine Synchronization for Fault Tolerance using DomT", Xen Summit 2008, Boston, MA.

[21] A.W. Appel and K. Li, "Virtual memory primitives for user programs" ASPLOS-IV Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, 1991.

[22] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm:Tracking processes in a virtual machine environment", In Proc. USENIX Annual Technical Conference, 2006.

[23] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment", In Proc. ASPLOS-XII, 2006.

[24] "Xen paravirt_ops for upstream Linux kernel", http://wiki.xensource.com/xenwiki/XenParavirtOps, Last accessed on April 12, 2011.

[25] Mark A. Weiss, "Data Structures and Algorithm Analysis," Second Edition, Addison Wesley.