Enhancing the Performance of High Availability Lightweight Live Migration

Peng Lu[1], Binoy Ravindran[1], and Changsoo Kim[2]

[1] ECE Department, Virginia Tech, USA
{lvpeng,binoy}@vt.edu
[2] ETRI, Daejeon, South Korea
cskim7@etri.re.kr

Abstract. Remus is one of the first systems which implemented whole virtual machine replication to achieve high availability (HA). Recently a fast, lightweight migration mechanism (LLM) was proposed to reduce the long network delay in Remus. However, these virtualized systems have the long downtime problem, which is a bottleneck to achieve HA. Based on LLM, in this paper, we describe a fine-grained block identification (or FGBI) mechanism to reduce the downtime in virtualized systems so as to achieve HA, with support for a block sharing mechanism and hybrid compression method. We implement the FGBI mechanism and evaluate it against LLM and Remus, using several benchmarks such as Apache, SPECweb, NPB and SPECsys. Our experimental results reveal that FGBI reduces the type I downtime over LLM and Remus by as much as 77% and 45% respectively, and reduces the type II downtime by more than 90% and more than 70%, compared with LLM and Remus respectively. Moreover, in all cases, the performance overhead of FGBI is less than 13%.

1 Introduction

High availability (HA) refers to a system and associated service implementation that is continuously operational for a long period of time. With respect to the clients, an ideal system never stops working, which also means the system will always respond to the clients' requests. Trying to achieve high availability is therefore one of the key concerns in modern cluster computing and failover systems. Whole-system replication is a conventional way to increase the system availability: once the primary machine fails, the running applications will be taken over by the backup machines. However, there are several limitations that make this method unattractive for deployment: it needs specialized hardware and software which are usually expensive. That the final system also requires complex customized configurations makes it hard to manage efficiently.

As virtualization becomes more and more prevalent, we can overcome these limitations by introducing the virtual machine (VM). In the virtual world, all the applications are running in the VM, so now it's possible to implement the whole-system replication in an easy and efficient way — by saving the copy of the whole VM running on the system. As VMs are totally hardware-independent,

A. Fernández Anta, G. Lipari, and M. Roy (Eds.): OPODIS 2011, LNCS 7109, pp. 50–64, 2011.

the cost is much lower compared to the hardware expenses in traditional HA solutions. Besides, virtualization technology can facilitate the management of multiple VMs on a single physical machine. With virtual machine monitors (VMM), the service applications are separated from physical machines, thus providing increased flexibility and improved performance.

Remus [6], built on top of the well-known Xen hypervisor [3], provides transparent, comprehensive high availability by using a checkpointing method under the Primary-Backup model (Figure 1). It checkpoints the running VM on the primary host, and transfers the latest checkpoint to the backup host as whole-system migration. Once the primary host fails, the backup host will take over the service based on the latest checkpoint. Remus proves that it is possible to create a general, fully transparent, high-availability solution entirely in software. However, checkpointing at high frequency will introduce significant overhead, since significant CPU and memory resources are consumed by the migration. Therefore, clients endure a long network delay.

Jiang *et. al.* [13] proposed an integrated live migration mechanism, called LLM, which integrates both whole-system checkpointing and input replay to reduce the network delay in Remus. The basic idea is that the primary host migrates the guest VM image (including CPU/memory status updates and new writes to the file system) to the backup host at low frequency. In the meanwhile, the service requests from network clients are migrated at high frequency. As its results show, LLM significantly outperforms Remus in terms of network delay by more than 90%.

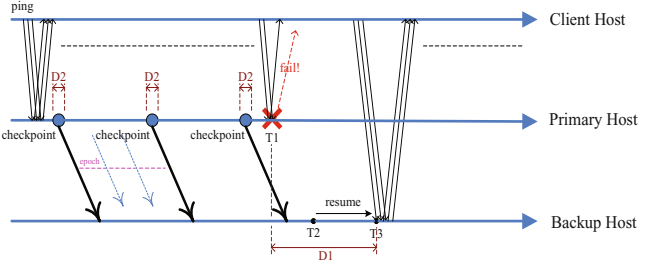


Fig. 1. Primary-Backup model and the downtime problem (T1: primary host crashes; T2: client host observes the primary host crashes; T3: VM resumes on backup host; D1 (T3 - T1): type I downtime; D2: type II downtime)

Downtime is the key factor for estimating the high availability of a system, since any long downtime experience for clients may result in loss of client loyalty and thus revenue loss. Under the Primary-Backup model, there are two types of downtime: I) the time from when the primary host crashes until the VM resumes from the last checkpointed state on the backup host and starts to handle client requests (shown as D1 in Figure 1); II) the time from when the VM pauses on the primary (to save

for the checkpoint) until it resumes (shown as D2 in Figure 1). From Jiang's paper we observe that for memory-intensive workloads running on guest VMs (such as the HighSys workload [13]), LLM endures much longer type I downtime than Remus. This is because, these workloads update the guest memory at high frequency. On the other side, LLM migrates the guest VM image update (mostly from memory) at low frequency but uses input replay as an auxiliary. In this case, when a failure happens, a significant number of memory updates are needed in order to ensure synchronization between the primary and backup hosts. Therefore, it needs significantly more time for the input replay process in order to resume the VM on the backup host and begin handling client requests.

Regarding the type II downtime, there are several migration epochs between two checkpoints, and the newly updated memory data is copied to the backup host at each epoch. At the last epoch, the VM running on the primary host is suspended and the remaining memory states are transferred to the backup host. Thus, the type II downtime depends on the amount of memory that remains to be copied and transferred when pausing the VM on the primary host. If we reduce the dirty data which need to be transferred at the last epoch, we can reduce the type II downtime. Moreover, if we reduce the dirty data which needs to be transferred at each epoch, trying to synchronize the memory state between primary and backup host all the time, then at the last epoch, there will not be too many new memory updates that need to be transferred, so we can reduce the type I downtime as well.

Therefore, in order to achieve HA in these virtualized systems, especially to address the downtime problem under memory-intensive workloads, we propose a memory synchronization technique for tracking memory updates, called Fine-Grained Block Identification (or FGBI). Our main contributions include:

1) Based on LLM, we develop a novel, efficient and fine-grained approach called FGBI, to track and transfer the memory updates efficiently, by reducing the total number of dirty bytes which need to be transferred from primary to backup host. FGBI enhances LLM's performance by overcoming its downtime disadvantage, especially for applications with memory-intensive workloads.

2) We integrate memory block sharing support with FGBI to reduce the newly introduced memory and computation/comparison overheads. In addition, we also support a hybrid compression mechanism among the memory dirty blocks to further reduce the migration traffic in the transfer period.

3) We present a fully functional prototype implementation and demonstrate that it achieves comparable downtime compared with Remus/LLM. Our experimental results reveal that FGBI reduces the type I downtime over LLM and Remus by as much as 77% and 45% respectively, and reduces the type II downtime by more than 90% and more than 70%, compared with LLM and Remus respectively.

The rest of the paper is organized as follows. Section 2 discusses past and related work. Section 3 presents the design and implementation of the integrated FGBI mechanism. Section 4 reports our experimental environment, benchmarks, and the evaluation results. We conclude and discuss future work in Section 5.

2 Related Work

To achieve high availability, currently there exist many virtualization-based live migration techniques [12, 18, 24]. Two representatives are Xen live migration [4] and VMware VMotion [17], which share similar pre-copy strategies. During migration, physical memory pages are sent from the source (primary) host to the new destination (backup) host, while the VM continues running on the source host. Pages modified during the replication must be re-sent to ensure consistency. After a bounded iterative transferring phase, a very short stop-and-copy phase is executed, during which the VM is halted, the remaining memory pages are sent, and the destination VMM is signaled to resume the execution of the VM. However, these pre-copy methods incur significant VM downtimes, as the evaluation results in [8] show.

Remus [6] is now part of the official Xen repository. It achieves HA by maintaining an up-to-date copy of a running VM on the backup host, which automatically activates if the primary host fails. Remus (and also LLM [13]) copies over dirty data after memory update, and uses the memory page as the granularity for copying. However, the dirty data tracking method is not efficient, as shown in [16] (we also illustrate this inefficiency in Section 3.1). Thus, our goal in this paper is to further reduce the size of the memory transferred from the primary to the backup host, by introducing a fine-grained mechanism.

Lu *et. al.* [16] applied three memory state synchronization techniques to achieve HA in systems such as Remus: dirty block tracking, speculative state transferring and active backup. The first technology is similar to our proposed method, however, it incurs additional memory associated overhead. For example, when running the Exchange workload in their evaluation, the memory overhead is more than 60%. Since main memory is always a scarce resource, the high percentage overhead is a problem. Different from these authors' work, we reduce memory overhead incurred by FGBI by integrating a new memory blocks sharing mechanism, and a hybrid compression method when transferring the memory update.

To solve the memory overhead problems under Xen-based systems, there are several ways to harness memory redundancy in VMs, such as page sharing and patching. Past efforts showed the memory sharing potential in virtualization-based systems. Working set changes were examined in [4, 21], and their results showed that changes in memory were crucial for the migration of VMs from host to host. For a guest VM with 512 MB memory assigned, low loads changed roughly 20 MB, medium loads changed roughly 80 MB, and high loads changed roughly 200 MB. Thus, normal workloads are likely to occur between these extremes. The evaluation in [4, 21] also revealed the amount of memory changes (within minutes) in VMs running different light workloads. None of them changed more than 4 MB of memory within two minutes. The Content-Based Page Sharing (CBPS) method [22] also illustrated the sharing potential in memory. CBPS was based on the compare-by-hash technique introduced in [9, 10]. As claimed, CBPS was able to identify as much as 42.9% of all pages as sharable, and reclaimed 32.9% of the pages from ten instances of Windows NT doing

real-world workloads. Nine VMs running Redhat Linux were able to find 29.2% of sharable pages and reclaimed 18.7%. When reduced to five VMs, the numbers were 10.0% and 7.2%, respectively.

To share memory pages efficiently, recently, the Copy-on-Write (CoW) sharing mechanism was widely exploited in the Xen VMM [19]. Unlike the sharing of pages within an OS that uses CoW in a traditional way, in virtualization, pages are shared between multiple VMs. Instead of using CoW to share pages in memory, we use the same idea in a more fine-grained manner, i.e., by sharing among smaller blocks. The Difference Engine project demonstrates the potential memory savings available from leveraging a combination of page sharing, patching, and in-core memory compression [8]. It shows the huge potential of harnessing memory redundancy in VMs. However, Difference Engine also suffers from complexity problems when applying the patching method. It needs additional modifications to Xen. We will present our corresponding mechanism and advantages over Difference Engine in Section 3.2.

Besides high availability systems such as Remus, LLM, and Kemari [20], which apply the pre-copy mechanism, there are also other related works that focus on migration optimization. Post-copy based migration [11] is proposed to address the drawbacks of pre-copy based migration. The experimental evaluation in [11] shows that the migration time using the post-copy method is less than the pre-copy method, under SPECweb2005 and Linux Kernel Compile benchmarks. However, its implementation only supports PV guests as the mechanism for trapping memory accesses and utilizes an in-memory pseudo-paging device in the guest OS. Since the post-copy mechanism needs to modify the guest OS, it is not so much widely used as the pre-copy mechanism.

3 Design and Implementation

We first overview the integrated FGBI design, including some necessary preliminaries about the memory saving potential. We then present the FGBI architecture, explain each component, and discuss the execution flow and other implementation details.

3.1 FGBI

Remus and LLM track memory updates by keeping evidence of the dirty pages at each migration epoch. Remus uses the same page size as Xen (for x86, this is 4KB), which is also the granularity for detecting memory changes. However, this mechanism is not efficient. For instance, no matter what changes an application makes to a memory page, even just modify a boolean variable, the whole page will still be marked dirty. Thus, instead of one byte, the whole page needs to be transferred at the end of each epoch. Therefore, it is logical to consider tracking the memory update at a finer granularity, like dividing the memory into smaller blocks.

We propose the FGBI mechanism which uses memory blocks (smaller than page sizes) as the granularity for detecting memory changes. FGBI calculates the hash value for each memory block at the beginning of each migration epoch. Then it uses the same mechanism as Remus to detect dirty pages. However, at the end of each epoch, instead of transferring the whole dirty page, FGBI computes new hash values for each block and compares them with the corresponding old values. Blocks are only modified if their corresponding hash values do not match. Therefore, FGBI marks such blocks as dirty and replaces the old hash values with the new ones. Afterwards, FGBI only transfers dirty blocks to the backup host.

However, because of using block granularity, FGBI introduces new overhead. If we want to accurately approximate the true dirty region, we need to set the block size as small as possible. For example, to obtain the highest accuracy, the best block size is one bit. That is impractical, because it requires storing an additional bit for each bit in memory, which means that we need to double the main memory. Thus, a smaller block size leads to a greater number of blocks and also requires more memory for storing the hash values. Based on these past efforts illustrating the memory saving potential (section 2), we present two supporting techniques: block sharing and hybrid compression. These are discussed in the subsections that follow.

3.2 Block Sharing and Hybrid Compression Support

From the memory saving results of related work (section 2), we observe that while running normal workloads on a guest VM, a large percentage of memory is usually not updated. For this static memory, there is a high probability that pages can be shared and compressed to reduce memory usage.

Block Sharing. Note that these past efforts [4, 9, 10, 21, 22] use the memory page as the sharing granularity. Thus, they still suffer from the "one byte differ, both pages cannot be shared" problem. Therefore, we consider using a smaller block in FGBI as the sharing granularity to reduce memory overhead.

The Difference Engine project [8] also illustrates the potential savings due to sub-page sharing, both within and across virtual machines, and achieves savings up to 77%. In order to share memory at the sub-page level, the authors construct patches to represent a page as the difference relative to a reference page. However, this patching method requires selected pages to be accessed infrequently, otherwise the overhead of compression/decompression outweighs the benefits. Their experimental evaluations reveal that patching incurs additional complexity and overhead when running memory-intensive workloads on guest VMs (from results for "Random Pages" workload in [8]).

Unlike Difference Engine, we apply a straightforward sharing technique to reduce the complexity. The goal of our sharing mechanism is to eliminate redundant copies of identical blocks. We share blocks and compare hash values in memory at runtime, by using a hash function to index the contents of every block. If the hash value of a block is found more than once in an epoch, there is a good probability that the current block is identical to the block that gave the same hash value. To ensure than these blocks are identical, they are compared

bit by bit. If the blocks are identical, they are reduced to one block. If, later on, the shared block is modified, we need to decide which of the original constituent blocks has been updated and will be transferred.

Hybrid Compression. Compression techniques can be used to significantly improve the performance of live migration [14]. Compressed dirty data takes shorter time to be transferred through the network. In addition, network traffic due to migration is significantly reduced when much less data is transferred between primary and backup hosts. Therefore, for dirty blocks in memory, we consider compressing them to reduce the amount of transferred data.

Before transmitting a dirty block, we check for its presence in an address-indexed cache of previously transmitted blocks (through pages). If there is a cache hit, the whole page (including this memory block) is XORed with the previous version, and the differences are run-length encoded (RLE). At the end of each migration epoch, we send only the delta from a previous transmission of the same memory block, so as to reduce the amount of migration traffic in each epoch. Since smaller amount of data is transferred, the total migration time and downtime can both be decreased.

However, in the current migration epoch, there still may remain a significant fraction of blocks that is not present in the cache. In these cases, we find that Wilson *et. al.* [7] claims that there are a great number of zero bytes in the memory pages (so as in our smaller blocks). For this kind of block, we just scan the whole block and record the information about the offset and value of nonzero bytes. And for all other blocks with weak regularities, a universal algorithm with high compression ratio is appropriate. Here we apply a general-purpose and very fast compression technique, zlib [1], to achieve a higher degree of compression.

3.3 Architecture

We implement the FGBI mechanism integrated with sharing and compression support, as shown in Figure 2. In addition to LLM, which is labeled as "LLM Migration Manager" in the figure, we add a new component, shown as "FGBI", and deploy it at both Domain 0 and guest VM.

For easiness in presentation, we divide FGBI into three main components:

1) Dirty Identification: It uses the hash function to compute the hash value for each block, and identify the new update through the hash comparison at the end of migration epoch. It has three subcomponents:
Block Hashing: It creates a hash value for each memory block;
Hash Indexing: It maintains a hash table based on the hash values generated by the Block Hashing component. The entry in the content index is the hash value that reflects the content of a given block;
Block Comparison: It compares two blocks to check if they are bitwise identical.

2) Block Sharing Support: It handles sharing of bitwise identical blocks.

3) Block Compression: It compresses all the dirty blocks on the primary side, before transferring them to the backup host. On the backup side, after receiving

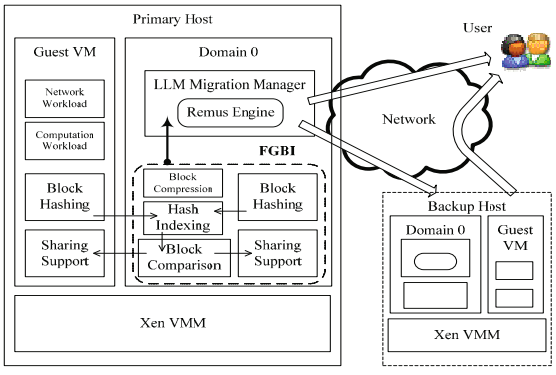


Fig. 2. The FGBI architecture with sharing and compression support

the compressed blocks, it decompresses them first before using them to resume the VM.

Basically, the Block Hashing component produces hash values for all blocks and delivers them to the Hash Indexing component. The Hash Indexing and Block Comparison components then check the hash table to determine whether there are any duplicate blocks. If so, the Hash Comparison component requests the Block Sharing Support components to update the shared blocks information. At the end of each epoch, the Block Compression component compresses all the dirty blocks (including both shared and not shared).

In this architecture, the components are divided between the privileged VM Domain 0 and the guest VMs. The VMs contain the Block Sharing Support components. We house the Block Sharing Support component in the guest VMs to avoid the overhead of using shadow page tables (SPTs). Each VM also contains a Block Hashing component, which means that it has the responsibility of hashing its address space. The Dirty Identification component is placed in the trusted and privileged Domain 0. It receives hash values of the hashed blocks generated by the Block Hashing component in the different VMs.

3.4 FGBI Execution Flow

Figure 3 describes the execution flow of the FGBI mechanism. The numbers on the arrows in the figure correspond to numbers in the enumerated list below:

1) Hashing: At the beginning of each epoch, the Block Hashing components at the different guest VMs compute the hash value for each block.

2) Storing: FGBI stores and delivers the hash key of the hashed block to the Hash Indexing component.

3) Index Lookup: It checks the content index for identical keys, to determine whether the block has been seen before. The lookup can have two different outcomes:

Key not seen before: Add it to the index and proceed to step 6.

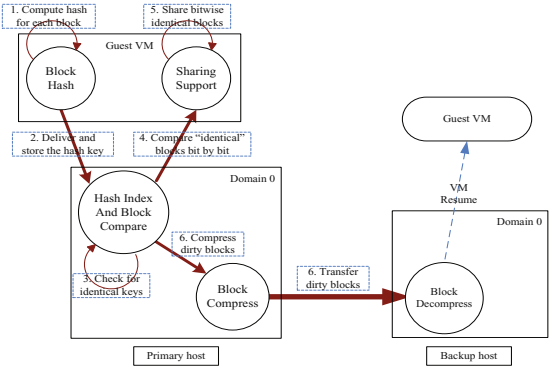


Fig. 3. Execution flow of FGBI mechanism

Key seen before: An opportunity to share, so request block comparison.

4) Block Comparison: Two blocks are shared if they are bitwise identical. Meanwhile, it notifies the Block Sharing Support Components on corresponding VMs that they have a block to be shared. If not, there is a hash collision, the blocks are not shared, and proceed to step 6.

5) Shared Block Update: If two blocks are bitwise identical, then store the same hash value for both blocks. Unless there is a write update to this shared block, it doesn't need to be compared at the end of the epoch.

6) Block Compression: Before transferring, compress all the dirty blocks.

7) Transferring: At the end of epoch, there are three different outcomes:

Block is not shared: FGBI computes the hash value again and compares with the corresponding old value. If they don't match, mark this block as dirty, compress and send it to the backup host. Repeat step 1 (which means begin the next migration epoch).

Block is shared but no write update: It means that either block is modified during this epoch. Thus, there is no need to compute hash values again for this shared block, and therefore, there is no need to make comparison, compression, or transfer either. Repeat step 1.

Block is shared and write update occurs: This means that one or both blocks have been modified during this epoch. Thus, FGBI needs to check which one is modified, and then compress and send the dirty one or both to the backup host. Repeat step 1.

4 Experimental Evaluation

We experimentally evaluated the performance of the proposed techniques (i.e., FGBI, sharing, and compression), which i simply referred to here as the FGBI mechanism. We measured downtime and overhead under FGBI, and compared the result with that under LLM and Remus.

4.1 Experimental Environment

Our experimental platform included two identical hosts (one as primary and the other as backup), each with an IA32 architecture processor (Intel Core 2 Duo Processor E6320, 1.86 GHz), and 3 GB RAM. We set up a 1 Gbps network connection between the two hosts, which is specifically used for migration. In addition, we used a separate machine as a network client to transmit service requests and examine the results based on the responses. We built Xen 3.4 from source [23], and let all the protected VMs run PV guests with Linux 2.6.18. The VMs were running CentOS Linux, with a minimum of services executing, e.g., sshd. We allocated 256 MB RAM for each guest VM, the file system of which is an image file of 3 GB shared by two machines using NFS. Domain 0 had a memory allocation of 1 GB, and the remaining memory was left free. The Remus patch we used was the nearest 0.9 version [5]. We compiled the LLM source code and installed its modules into Remus.

Our experiments used the following VM workloads under the Primary-Backup model:

Static web application: We used Apache 2.0.63 [15]. Both hosts were configured with 100 simultaneous connections, and repetitively downloaded a 256KB file from the web server. Thus, the network load will be high, but the system updates are not so significant.

Dynamic web application: SPECweb99 is a complex application-level benchmark for evaluating web servers and the systems that host them. This benchmark comprises a web server, serving a complex mix of static and dynamic page (e.g., CGI script) requests, among other features. Both hosts generate a load of 100 simultaneous connections to the web server [2].

Memory-intensive application: Since FGBI is proposed to solve the long downtime problem under LLM especially when running heavy computational workloads on the guest VM, we continued our evaluation by comparing FGBI with LLM/Remus under a set of industry-standard workloads, specifically NPB and SPECsys.

1. *NPB-EP:* This benchmark is derived from CFD codes, and is a standard measurement procedure used for evaluating parallel programs. We selected the Kernel EP program from the NPB benchmark [19], because the scale of this program set is moderate and its memory access style is representative. Therefore, this example involves high computational workloads on the guest VM.

2. *SPECsys:* This benchmark measures NFS (version 3) file server throughput and response time for an increasing load of NFS operations (lookup, read, write, and so on) against the server over file sizes ranging from 1 KB to 1 MB. The page modification rate when running SPECsfs has previously been reported as approximately 10,000 dirty pages/second [2], which is approximately 40% of the link capacity on a 1 Gbps network.

To ensure that our experiments are statistically significant, each data point is averaged from twenty sample values. The standard deviation computed from the samples is less than 7.6% of the mean value.

4.2 Downtime Evaluations

Type I Downtime. Figures 4a, 4b, 4c, and 4d show the type I downtime comparison among FGBI, LLM, and Remus mechanisms under Apache, NPB-EP, SPECweb, and SPECsys applications, respectively. The block size used in all experiments is 64 bytes. For Remus and FGBI, the checkpointing period is the time interval of system update migration, whereas for LLM, the checkpointing period represents the interval of network buffer migration. By configuring the same value for the checkpointing frequency of Remus/FGBI and the network buffer frequency of LLM, we ensure the fairness of the comparison. We observe that Figures 4a and 4b show a reverse relationship between FGBI and LLM. Under Apache (Figure 4a), the network load is high but system updates are rare. Therefore, LLM performs better than FGBI, since it uses a much higher frequency to migrate the network service requests. On the other hand, when running memory-intensive applications (Figures 4b and 4d), which involve high computational loads, LLM endures a much longer downtime than FGBI (even worse than Remus).

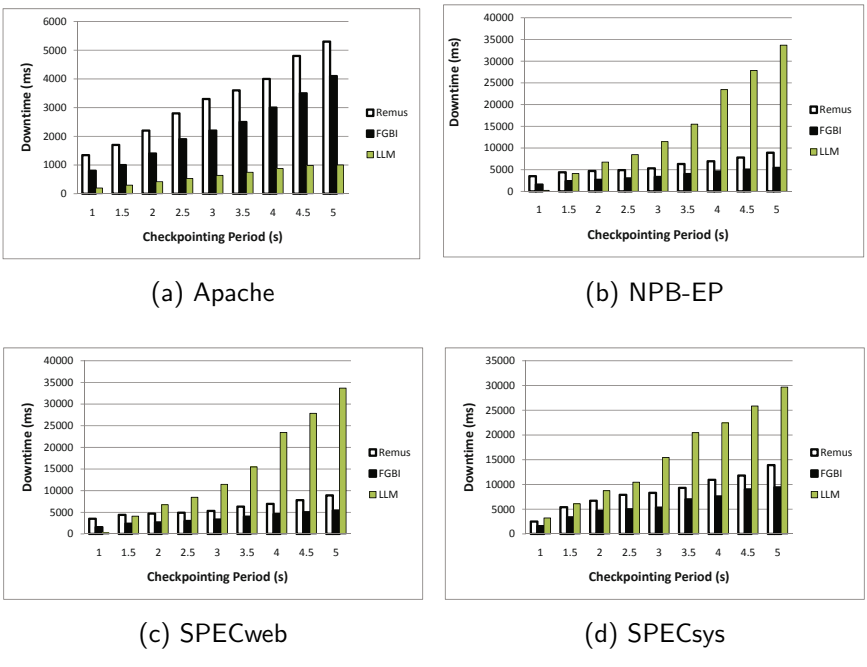


Fig. 4. Type I downtime comparison under different benchmarks

Although SPECweb is a web workload, it still has a high page modification rate, which is approximately 12,000 pages/second [4]. In our experiment, the 1 Gbps

migration link is capable of transferring approximately 25,000 pages/second. Thus, SPECweb is not a lightweight computational workload for these migration mechanisms. As a result, the relationship between FGBI and LLM in Figure 4c is more similar to that in Figure 4b (and also Figure 4d), rather than Figure 4a. In conclusion, compared with LLM, FGBI reduces the downtime by as much as 77%. Moreover, compared with Remus, FGBI yields a shorter downtime, by as much as 31% under Apache, 45% under NPB-EP, 39% under SPECweb, and 35% under SPECsys.

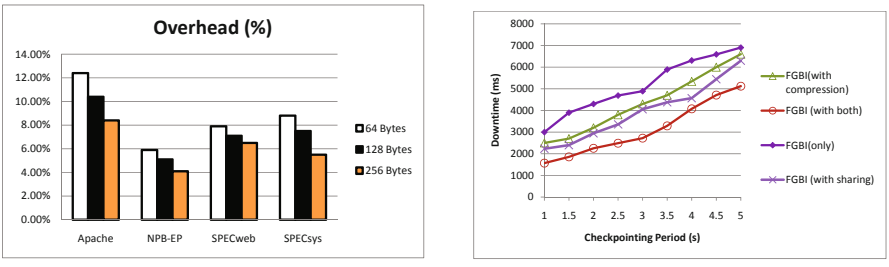
Type II Downtime. Table 1 shows the type II downtime comparison among Remus, LLM, and FGBI mechanisms under different applications. We have three main observations: (1) Their downtime results are very similar for the idle run. This is because, Remus is a fast checkpointing mechanism and both LLM and FGBI are based on it. Memory updates are rare during the "idle" run, so the type II downtime in all three mechanisms is short. (2) When running the NPB-EP application, the guest VM memory is updated at a high frequency. When saving the checkpoint, LLM takes much more time to save huge dirty data caused by its low memory transfer frequency. Therefore, in this case FGBI achieves a much lower downtime than Remus (more than 70% reduction) and LLM (more than 90% reduction). (3) When running the Apache application, the memory update is not so much as that when running NPB, but the memory update is definitely more than idle run. The downtime results shows that FGBI still outperforms both Remus and LLM.

Table 1. Type II downtime comparison

Application	Remus downtime	LLM downtime	FGBI downtime
idle	64 ms	69 ms	66 ms
Apache	1032 ms	687 ms	533 ms
NPB-EP	1254 ms	16683 ms	314 ms

4.3 Overhead Evaluations

Figure 5a shows the overhead during VM migration. The figure compares the applications' runtime with and without migration, under Apache, SPECweb, NPB-EP, and SPECsys, with the size of the fine-grained blocks varying from 64 bytes to 128 bytes and 256 bytes. We observe that in all cases the overhead is low, no more than 13% (Apache with 64 bytes block). As discussed in Section 3, the smaller the block size that FGBI chooses, greater is the memory overhead that it introduces. In our experiments, the smaller block size that we chose is 64 bytes, so this is the worst case overhead compared with the other block sizes. Even in this "worst" case, under all these benchmarks, the overhead is less than 8.21%, on average.



(a) Overhead under different block size. (b) Comparison of proposed techniques.

Fig. 5. Overhead Measurements

In order to understand the respective contributions of the three proposed techniques (i.e., FGBI, sharing, and compression), Figure 5b shows the breakdown of the performance improvement among them under the NPB-EP benchmark. It compares the downtime between integrated FGBI (which we use for evaluation in this Section), FGBI with sharing but no compression support, FGBI with compression but no sharing support, and FGBI without sharing nor compression support, under the NPB-EP benchmark. As previously discussed, since NPB-EP is a memory-intensive workload, it should present a clear difference among the three techniques, all of which focus on reducing the memory-related overhead. We do not include the downtime of LLM here, since for this compute-intensive benchmark, LLM incurs a very long downtime, which is more than 10 times the downtime that FGBI incurs.

We observe from Figure 5b that if we just apply the FGBI mechanism without integrating sharing or compression support, the downtime is reduced, compared with that of Remus in Figure 4b, but it is not significant (reduction is no more than twenty percent). However, compared with FGBI with no support, after integrating hybrid compression, FGBI further reduces the downtime, by as much as 22%. We also obtain a similar benefit after adding the sharing support (downtime reduction is a further 26%). If we integrate both sharing and compression support, the downtime is reduced by as much as 33%, compared to FGBI without sharing or compression support.

5 Conclusions

One of the primary bottlenecks on achieving high availability in virtualized systems is downtime. We presented a novel fine-grained block identification mechanism, called FGBI, that reduces the downtime in lightweight migration systems. In addition, we developed a memory block sharing mechanism to reduce the memory and computational overheads due to FGBI. We also developed a dirty block compression support mechanism to reduce the network traffic at each migration epoch. We implemented FGBI with the sharing and compression mechanisms and integrated them with the LLM lightweight migration system. Our

experimental evaluations reveal that FGBI overcomes the downtime disadvantage of LLM by more than 90%, and of Xen/Remus by more than 70%. In all cases, the performance overhead of FGBI is less than 13%.

Several directions for future work exist. It is possible to reduce the implementation complexity in the FGBI design. For instance, we can deploy some subcomponents (such as the Block Comparison part) in the VMM directly, and design a transparent solution by using the shadow page table mechanism. Moreover, compressing memory blocks that are unlikely to be accessed in the near future can further reduce the memory overhead.

Acknowledgment. We thank the anonymous reviewers for their very helpful comments. This work was supported by the IT R&D program of MKE/KEIT/ETRI, South Korea [K1001703, A Development of Cost Effective and Large Scale Global Internet Service Solution].

References

1. http://www.zlib.net
2. Akoush, S., Sohan, R., Rice, A., Moore, A.W., Hopper, A.: Predicting the performance of virtual machine migration. In: International Symposium on Modeling, Analysis, and Simulation of Computer Systems, pp. 37–46 (2010)
3. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 164–177. ACM, New York (2003)
4. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, NSDI 2005, vol. 2, pp. 273–286. USENIX Association, Berkeley (2005)
5. Cully, B., Lefebvre, G., Hutchinson, N., Warfield, A.: Remus source code, http://dsg.cs.ubc.ca/remus/
6. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: high availability via asynchronous virtual machine replication. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2008, pp. 161–174. USENIX Association, Berkeley (2008)
7. Ekman, M., Stenstrom, P.: A robust main-memory compression scheme. In: ISCA 2005: Proceedings of the 32nd Annual International Symposium on Computer Architecture, pp. 74–85. IEEE Computer Society, Washington, DC (2005)
8. Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A.: Difference engine: harnessing memory redundancy in virtual machines. Commun. ACM 53, 85–93 (2010)
9. Henson, V.: An analysis of compare-by-hash. In: Proceedings of the 9th Conference on Hot Topics in Operating Systems, vol. 9, pp. 3–3 (2003)
10. Henson, V., Henderson, R.: Guidelines for using compare-by-hash, http://infohost.nmt.edu/~val/review/hash2.pdf
11. Hines, M.R., Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2009, pp. 51–60. ACM, New York (2009)

12. Huang, W., Gao, Q., Liu, J., Panda, D.K.: High performance virtual machine migration with RDMA over modern interconnects. In: CLUSTER 2007: Proceedings of the 2007 IEEE International Conference on Cluster Computing, pp. 11–20. IEEE Computer Society, Washington, DC (2007)
13. Jiang, B., Ravindran, B., Kim, C.: Lightweight Live Migration for High Availability Cluster Service. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 420–434. Springer, Heidelberg (2010)
14. Jin, H., Deng, L., Pan, X.: Live virtual machine migration with adaptive, memory compression. In: 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1–10 (2009)
15. Liu, H., Jin, H., Liao, X., Hu, L., Yu, C.: Live migration of virtual machine based on full system trace and replay. In: HPDC 2009: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, pp. 101–110. ACM, New York (2009)
16. Lu, M., Chiueh, T.C.: Fast memory state synchronization for virtualization-based fault tolerance. In: Dependable Systems Networks, pp. 534–543 (2009)
17. Nelson, M., Lim, B.H., Hutchins, G.: Fast transparent migration for virtual machines. In: ATEC 2005: Proceedings of the Annual Conference on USENIX Annual Technical Conference, pp. 25–25. USENIX Association, Berkeley (2005)
18. Bradford, A.F.R., Kotsovinos, E., Schioeberg, H.: Live wide-area migration of virtual machines including local persistent state. In: VEE 2007: Proceedings of the Third International Conference on Virtual Execution Environments, pp. 169–179. ACM Press, San Diego (2007)
19. Sun, Y., Luo, Y., Wang, X., Wang, Z., Zhang, B., Chen, H., Li, X.: Fast live cloning of virtual machine based on Xen. In: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications, pp. 392–399. IEEE Computer Society, Washington, DC (2009)
20. Tamura, Y., Sato, K., Kihara, S., Moriai, S.: Kemari: Virtual machine synchronization for fault tolerance using DomT (technical report). `http://wiki.xen.org/xenwiki/Open_Topics_For_Discussion?action=AttachFile&do=get&target=Kemari_08.pdf` (June 2008)
21. Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A.C., Voelker, G.M., Savage, S.: Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 148–162. ACM, New York (2005)
22. Waldspurger, C.A.: Memory resource management in vmware esx server. SIGOPS Oper. Syst. Rev. 36, 181–194 (2002)
23. XenCommunity. Xen unstable source, `http://xenbits.xensource.com/xen-unstable.hg`
24. Zhao, M., Figueiredo, R.J.: Experimental study of virtual machine migration in support of reservation of cluster resources. In: VTDC 2007: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, pp. 5:1–5:8. ACM, New York (2007)