Optimizing Post-Copy Live Migration with System-Level Checkpoint Using Fabric-Attached Memory

Chih Chieh Chou*[1], Yuan Chen†[2], Dejan Milojicic‡, A. L. Narasimha Reddy*, and Paul V. Gratz*
*Department of Electrical and Computer Engineering
Texas A&M University
Email: {ccchou2003, reddy, pgratz}@tamu.edu
†JD.com Silicon Valley R&D Center
Email: yuan.chen@jd.com
‡Hewlett Packard Labs
Email: dejan.milojicic@hpe.com

***Abstract*—Emerging Non-Volatile Memories have byte-addressability and low latency, close to the latency of main memory, together with the non-volatility of storage devices. Similarly, recently emerging interconnect fabrics, such as Gen-Z, provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs). FAMs can serve to create scalable, high-bandwidth, distributed, shared, byte-addressable, and non-volatile memory pools at a rack scale, opening up new usage models and opportunities.**

Based on these attractive properties, in this paper we propose FAM-aware, checkpoint-based, post-copy live migration mechanism to improve the performance of migration. We have implemented our prototype with a Linux open source checkpoint tool, CRIU (Checkpoint/Restore In Userspace). According to our evaluation results, compared to the existing solution, our FAM-aware post-copy can improve at least 15% the total migration time, at least 33% the busy time, and can let the migrated application perform at least 12% better during migration.

I. Introduction

The emerging Non-Volatile Memories (NVM), such as phase-change memory (PCM) [1], NVDIMM [2], and 3D XPoint [3], have byte-addressability and low latency, close to that of main memory, together with the non-volatility of storage devices. Similarly, recently emerging interconnect fabrics, such as Gen-Z [4], provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs) [5]. FAMs can serve to create scalable, high-bandwidth, distributed, shared, and byte-addressable NVM pools at a rack scale, opening up new usage models and opportunities. These technologies have great potential to improve the system/application performance as well as to provide scalability and reliability of applications/service. Many prior work focused on new system designs using NVM [6]–[11] and has already shown promising performance improvements.

Migration is a crucial technique for load balancing. Migration allows system administrators to remove some applications from stressed physical nodes in order to redistribute load, and therefore to increase overall system performance. In addition, migration can also provide power saving [12], and online maintenance. Traditional approaches to migration are non-live; that is, they require the application to be taken off-line while the migration occurs. Non-live migration can be divided into three steps: 1) the program is checkpointed at source, 2) the checkpointed data are copied from source to target, and 3) the program is restarted at target. The main drawback of non-live migration is that its application downtime (off-line time) is too long. To reduce this downtime, live migrations [13] are proposed to migrate most of (or all) pages before (pre-copy) or after (post-copy) "real" migration, and therefore to reduce the number of migrated pages during the downtime. However, the side effect of live migrations is that they require longer, compared to non-live migration, busy time of source node.

Here, we define the busy time as the duration from the beginning of the migration to the time that migrated applications can be killed at source node.[1] As far as we know, all prior works of post-copy focused on total migration time. However, we would like to make another point here that the busy time might be more important because it has direct impacts on the system performance; it decides when computing resources, such as CPU and memory, occupied by migrated applications to be released and be reallocated to remaining applications in source node. For example, when applications are suffering from swapping due to the lack of memory in a "hot" node, simply migrating an application with large memory footprint might be able to stop (after busy time) all remaining applications at source from swapping and therefore to improve the

[1] The author started this work as an intern at Hewlett Packard Labs.
[2] The author contributed while he was at Hewlett Packard Labs.

[1]Note: after migration completes, the migrated applications would continue to execute at target node.

978-1-7281-6007-8/19/$31.00 ©2019 IEEE
DOI 10.1109/MCHPC49590.2019.00010

overall system performance.

In this work, we consider that migration can greatly benefit from FAM. Although the non-live migration techniques can be easily improved with FAM by the removal of the copy phase (step 2), the optimal post-copy page fault handler, however, requires significant redesign. This new handler should rely on the both non-volatility and shareability of FAM to provide the optimal (shortest) busy time as well as total migration time. In particular, we introduce FAM-aware, checkpoint-based, post-copy live migration, which checkpoints the entire application to FAM in the beginning and transfers all pages through FAM, rather than through the network connection, which the traditional migration techniques use.

The contributions of this paper are as follows:

- Propose a new, checkpoint-based, page fault handler for post-copy migration using FAM to provide the best busy time and better total migration time.
- Implement our FAM-aware post-copy migration on a Linux open source checkpointing tool, CRIU (Checkpoint/Restore in Userspace).
- Evaluate our enhancements of CRIU with synthetic and realistic (YCSB+REDIS) workloads and show significant performance improvement.

The remainder of this paper is organized as follows. Section II describes the background and related work. Section III presents the motivation and design overview of our FAM-aware post-copy live migration. Section IV explains the implementation of our work by modifying and enhancing existing CRIU in more detail. Section V presents our results of evaluation of post-copy live migration using some macro benchmarks and realistic workloads. Finally, section VI concludes.

II. Background

A. Migration

Traditionally, in non-live migration of applications, at first the migration program needs to stop the applications. It then checkpoints their state (mostly as files) into local storage devices, copies these checkpointed data to the remote storage devices (at the target machine), and finally resumes them back to the checkpointed state at target. The downtime is mostly proportional to the amount of migrated memory.

Unlike non-live migration, live migration means that application is (or appears to be) responsive during the entire migration process. Post-copy [14], [15] transfers the processor state, register, etc., to target and resumes immediately at the target host. When resumed application accesses some pages which have not yet been transferred, page faults are triggered, and those pages are retrieved from source node through network.

Although post-copy has the almost minimum downtime, it would suffer from the network page fault overhead, and therefore degrade the migrated application performance during migration. Also, post-copy usually requires the longer migration time (which depends on page access pattern of application), compared with non-live one.

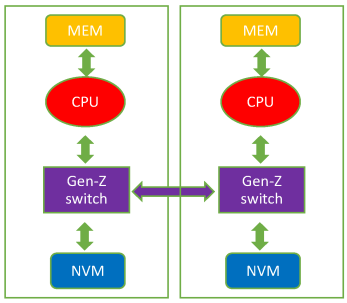


Fig. 1: Fabric-Attached Memory.

Sahni et al. [16] proposed a hybrid approach combining pre-copy and post-copy to take advantage of both types of live migration. The post-copy could suffer less page faults if the pages of the working set have already been sent by the pre-copy phase. Ye et al. [17] investigated the migration efficiency under different resource reservation strategies, such as CPU and memory reserved at target or source nodes. CQNCR [18] considered an optimal migration plan to migrate massive virtual machines in the data center by deciding a migration order to have less migration time and system impact.

These works, however, largely employ traditional networking as the only transfer media, thus they incur high access latencies. Besides, they only emphasize the total migration time, not busy time.

B. Fabric-Attached Memory

Fabric-Attached Memory (FAM) is a system architecture component in which high performance networking fabrics are used to interconnect NVM between multiple nodes in a rack to create a global, shared NVM pool. In our system model, we consider a cluster consisting of many nodes, each of which contains both DRAM and NVM. DRAM can only be accessed locally by local memory controller and serves as fast, local memory in each node. NVM and the processor (in each node) are connected to a switch fabric and these switches are interconnected to each other. Here we focus on Gen-Z [4] as one such fabric because it supports hundreds gigabyte per second bandwidth and memory semantics; however, any other fabric of similar latency and bandwidth could be used. The CPU can access NVM at other nodes with memory-semantics through the fabric interface and libraries. Therefore, all interconnected NVM can be treated as the slow, global memory pool in a rack scale [5]. This byte-addressable, shared, high-bandwidth, global NVM pool is so-called Fabric-Attached Memory (FAM) in this work. Fig. 1 shows the overall memory, NVM, CPU, and Gen-Z interconnections.

C. Checkpoint/Restore in Userspace (CRIU)

The Checkpoint/Restore in Userspace (CRIU) [19] is a Linux open source checkpoint tool which saves the current state of a running application into the local storage devices and restarts the application whenever necessary. To checkpoint, CRIU uses `ptrace` system call to inject a piece of parasite

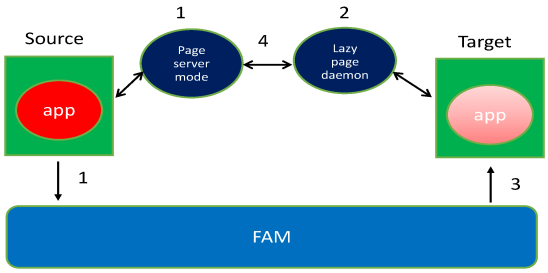


Fig. 2: Existing CRIU post-copy using FAM. (1) CRIU checkpoints all files except for page image file to FAM, and transforms itself as a page-server. (2) At target, CRIU creates a lazy-page daemon. (3) After (1) is completed, CRIU restores application immediately at target. (4) If restored application at target accesses a page and causes a page fault, it would notify the lazy-page daemon, which then requests to and obtains from page-server at source that faulting page.

code into the checkpointed application. Through the injected parasite, CRIU daemonizes the checkpointed application and lets it begin to accept commands sent by CRIU. Hereafter, CRIU starts the checkpoint process.

The most time-consuming part of checkpoint process is to dump pages of application. The page-dumping request asks the application to execute `vmsplice` system call, which maps the pages of VMAs of the process into pipes (kernel buffers). Finally, after all pages have already been mapped to the pipes, CRIU can access them directly (without the help of parasite) through `splice` system call, which copies dumped pages from pipes to a page image file in the storage devices.

CRIU also supports migration features, including non-live, pre-copy, and post-copy live migrations. For post-copy, the page fault handling is the main challenge. Like on-demand paging used in virtual memory systems, CRIU's post-copy employs the `userfaultfd` system call to allow paging in the user space. Fig. 2 shows the sequential steps of the existing CRIU post-copy implementation. To achieve the post-copy, like checkpoint, at first CRIU maps the pages of VMAs of applications, at the source node, into pipes and then places itself into a page-server mode. Then a lazy-page daemon at the target node is created to handle the page fault and other events requested during the following restore operations. Finally, the migrated application is resumed at the target node. When the restored application accesses a page which still remains in the source, the application is halted temporarily, and sends the page fault request to the lazy-page daemon, which in turn communicates with page-server at the source node to obtain that faulting page from pipes through network transfer. Although all other checkpointed state can be sent through FAM. The page transfer still needs to rely on socket interface if page fault handling is not redesigned.

We note that when using pure on-demand paging, migration process may never complete, since some pages may not be ever accessed. CRIU thus employs a timer to trigger the active pushing; that is, sequentially dumping all remaining pages from source to target. The timer is kept reset whenever a page fault event happens. Before the timer is expired, the lazy-page daemon only handles the page fault event, and it would start to actively push all remaining pages only after expiration.

III. Motivation and System Design Overview

In this section, we describe the motivation and design overview of our FAM-aware post-copy live migration.

A. Fabric-Attached Memory-Aware Post-Copy Live Migration

We propose an optimized FAM-aware post-copy migration, which exploits the properties of FAM to achieve low application downtime, low total migration time, low application degradation, low resume time, and especially low busy time.

To simplify the understanding of readers, we first briefly restate some key metrics of live migration proposed by Hines et al. [15] and we further introduce the concept of busy time.

- **Downtime:** The time that the application is stopped and cannot respond while the state of the processors and some pages are transferred. Post-copy, depending on implementation, might transfer a few (or no) pages. Non-live migration transfers all pages here, so its downtime is the longest.
- **Resume Time:** The time from the application starts executing to the end the entire migration process. This time is mainly required for post-copy to handle the page faults happening at the target node, and is near negligible to the non-live migration.
- **Busy Time:** The time from the beginning of the migration to the time that migrated applications can be killed and their resources (especially CPU and memory) can be released at source node. This may be the most important metric for migration since system administrators can only alleviate the loading of "hot" nodes after this time.
- **Migration Time:** The total time of downtime and resume time. Usually the migration time (of live migration) equals to the busy time; however, this is not the case if we employ FAM for migration. We will discuss this later.
- **Application Degradation:** The extent that application is slowed down during the operations of the migration. The application degradation of post-copy, because it handles the page fault at target and needs to get the faulting pages from source, might be the most severe.

B. Motivation and Design

Intuitively, the performance of non-live migration, especially migration time, could benefit from adopting the FAM simply because the copy phase can be eliminated through direct FAM accesses. Therefore, the entire migration process is now simplified as (1) checkpoint application to FAM at source and (2) restart application at target.

Post-copy migration is much more complicated than non-live one because the most critical part of post-copy is page fault handling (or network fault as termed by Hines et al. [15]) in the target. The behavior and implementation of the page

fault handler will greatly impact the resume time (application performance degradation) and busy time. Simply removing the copy phase, if applicable, is obviously not good enough. Therefore, our work focuses on optimizing FAM-aware post-copy migration based on the following three guidelines.

Checkpoint-Based Migration: The most apparent drawback of non-live migration is its very long downtime (which equals to the total migration time). However, the busy time of non-live migration is the best (compared to the live migrations) and is also much shorter than its total migration time because non-live migration first checkpoints everything to storage devices, and therefore, since a complete snapshot is saved in persistent media, the checkpointed application can be killed at the source.

Traditional live migrations have a long busy time (which equals to the total migration time) because they utilize memory to temporarily store pages and transfer pages by network. So, killing the application must wait until the completion of entire migration. Otherwise, if target crashes before migration completes, then the application will crash, too.

Due to the low-latency and non-volatility of FAM, storing migrated pages to FAM directly only slightly impacts performance (compared to DRAM), but it also saves the entire migrated state to persistent media. Therefore, application can be killed after being checkpointed and its busy time could be very close to that of non-live migration.

Accessing FAM as Shared Memory: Most existing live migration techniques [16]–[18], [20] still migrate their data content through communication network because they do not have a shared memory across multiple nodes. Therefore, their approaches will suffer the network overhead and network bandwidth. On the other hand, with the help of FAM, serving as a shared memory pool within the same rack, data could be simply migrated through memory semantics (`load/store` instructions), which bypass the significant networking protocol overhead. This could improve the critical page fault latency and therefore resume time and application degradation.

Retrieving Faulting Pages Synchronously: The page fault handler of the existing shared memory within a machine is controlled by the central operating system. The OS only needs to setup the page table of each process, then faulting pages can be mapped to virtual space of processes correctly. Our migration scheme, however, differs because a central controlling OS across multiple hosts does not exist. The FAM across machines can only be employed as a connecting media between nodes.

Besides, our page fault handler must contain two steps: first pages are written from source to FAM and then pages are read from FAM to target. So, page fault handling must be executed asynchronously through communication between the source (writing to FAM) and target nodes (reading from FAM); that is, the faulting address of pages must be sent to source first and then target must wait for the response to read that page. This communication impacts the page fault latency (even though all pages are already transferred by FAM) as well as migration performance, and must be avoided as much as possible by leveraging the information of the dumped pages. The source could notify target of the information of all dumped pages, and therefore the following faulting pages (if they have been dumped to FAM) can be accessed synchronously at target from FAM without the need of communication.

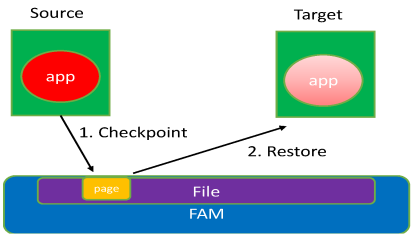


Fig. 3: Ideal migration method using FAM.

Thus, our post-copy optimization is mainly based on non-volatility and shared-ability of FAM, and can be divided into three parts.

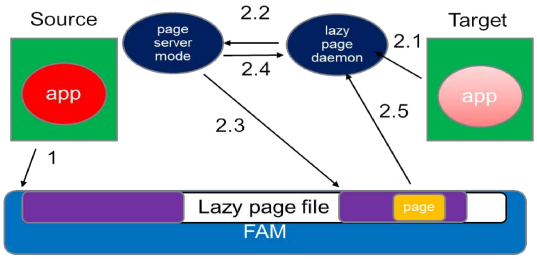
- *to achieve the shortest downtime*, we checkpoint the processor state, registers, etc., (excluding pages of VMA) of the victim application to FAM and resume victim application at the target.
- *to achieve low busy time*, at source, after checkpointing the necessary information, all the remaining pages continue to be dumped to FAM on the background. The victim application can be killed right after the page dumping is finished. This provides near optimal busy time, which is the checkpoint time of non-live migration.
- *to achieve low resume time and low application degradation*, all faulting pages at target will be served/received from FAM directly and the need of asynchronous communication is also tried to minimize. Therefore, with of help of FAM, the networking overhead as well as the page fault latency can be reduced significantly.

IV. Implementation

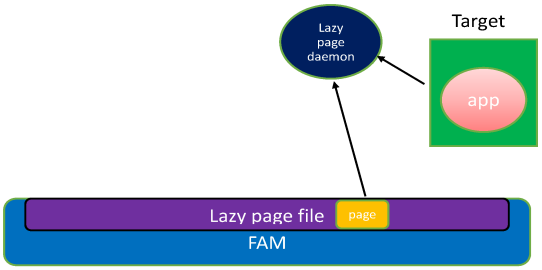
In this section, we explain our implementation of FAM-aware, checkpoint-based, post-copy live migration in detail.

The existing CRIU [19] migration tool is used as a baseline implementation, and augmented with our FAM-aware post-copy technique in this work. In particular, our case study implementation is based on modifying CRIU-dev branch version 3.2. Although our implementation is based upon CRIU, the technique developed is universal and may be easily implemented in other migration schemes.

Fig. 3 shows an ideal migration between nodes with FAM. First, an empty file is created in FAM, and the migration program at target can `mmap` this whole file and directly read dumped pages without having to wait until the whole file is dumped from the source. Ideally, (if the future can be predicted,) the source node would write a certain page to FAM each time before the page is needed at target. From the perspective of migration, which means that not only the restored application does not have to wait for the completion of page-dumping (that is, live migration), but it also avoids



(a) 1. Background thread dumps all lazy pages to FAM. 2.1. A page fault happens. A page fault event is sent to lazy-page daemon by kernel. 2.2. Lazy-page daemon requests page-server for that faulting page. 2.3. Page-server writes the entire vector, rather than a single faulting page, to FAM. 2.4. Page-server notifies the lazy-page daemon of the completion of dumping. 2.5. Lazy-page daemon directly reads faulting page from FAM.



(b) The page-server and application at source are killed after all lazy pages have been dumped. All remaining pages can be directly/synchronously accessed from target.

Fig. 4: FAM-aware post-copy live migration.

much of the network transmission (required by the existing CRIU and other previous implementations) through memory-semantics.

We have implemented the concept shown in Fig. 3 in our optimized post-copy migration in CRIU. Fig. 4 illustrates the main difference between our post-copy design versus the existing CRIU implementation (in Fig. 2).

Like the existing CRIU post-copy, we also employ a lazy-page daemon and page-server mode in our implementation. To migrate, first all required data are checkpointed as files to FAM except for the page image file, as the existing CRIU does. After that, CRIU at source enters a page-server mode, and launches a background thread to dump (checkpoint) all remaining "lazy" pages to FAM as a single lazy-page file (per process) (arrow with number 1 in Fig. 4(a)). Without having to wait for this background thread to finish its dumping job, system administrator can launch a lazy-page daemon and then can begin to restore the migrated application at the target node. To improve the checkpoint performance by batching and to avoid too long critical latency of page fault, we partition the VMAs of application as several I/O vectors with the maximum size of 2MB. Each I/O vector contains pages with contiguous virtual addresses. Therefore, the background thread writes the pages to FAM with at most 2MB of data at a time. When the restored application encounters a page fault, it sends the page fault event to the lazy-page daemon, which in turn sends page fault request to page-server. If the faulting page has not been dumped to FAM, the page-server waits for background thread to finish the dumping of current I/O vector, stops the background thread (by a spin lock), and dumps a specific I/O vector containing the requested faulting page. After dumping that (2MB) I/O vector, the page-server acknowledges page fault request and resumes the background thread. Arrows with number 2.1 to 2.4 in Fig. 4(a) illustrate this process. Alternately, if the faulting page has been dumped, the page-server acknowledges immediately.

The responses (arrow with number 2.4 in Fig. 4(a)) sent by the page-server not only indicate the "completion of dumping" of faulting pages, but they also contain some extra information for lazy-page daemon: the background thread's dumping progress (the largest virtual address of dumped pages) and the virtual address space of this entire (2MB) dumped I/O vector. The lazy-page daemon employs such information to construct a lookup table. (Actually, we build an LRU linked list). If the following faulting pages whose addresses can be found at the lookup table or are smaller than the dumping progress, they can be read from FAM synchronously without requesting to page-server. This can eliminate a lot of communications and overheads between source and target.

Once the background thread dumps all the lazy pages, the page-server actively notifies the lazy-page daemon of the completion of the checkpoint. Hereafter, the lazy-page daemon can synchronously read all faulting pages from FAM without any communication with page-server. Both migrated application and page-server can be killed at the source now as Fig. 4(b) shows.

Our implementation has some advantages: (1) The page-server only needs to handle faulting pages until the checkpoint of all "lazy" pages is finished. After that, all pages can be synchronously read from FAM. A lot of network transmissions of pages and protocol overhead can be eliminated. (2) The information of dumped pages is utilized to further reduce the communication between target and source nodes. It significantly minimizes the page fault latency and therefore improves performance. (3) After the background thread finishes dumping the lazy files, the application and page-server can be killed and their resources can be released; that is, the busy time would be much shorter than total migration time.

V. Evaluation

In this section, we experimentally evaluate the performance improvement of our optimized FAM-aware, checkpoint-based, post-copy live migration. Our platform contains 20GB DDR3-1600, 12GB NVM (emulated with DRAM [21]), and Intel i7-4770 four-core 3.4 GHz processor with hyperthreading enabled. Linux 4.15.0 is used in our platform.

A. Workloads and Experimental Setup

To examine the performance of our FAM-aware post-copy migration scheme, we leverage the NAS Parallel benchmarks (NPB) 3.3.1 [22] Class C, PARSEC 3.0 [23] native input, and

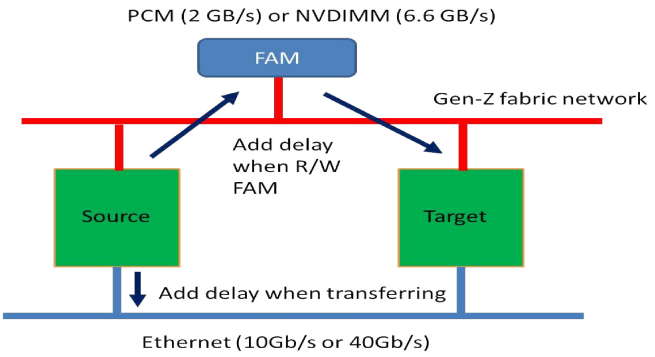


Fig. 5: The delay model of our evaluation.

TABLE I: Parameters for extra delay.

Media	Read lat. (us)	Write lat. (us)	BW. (GB/s)
PCM	1	1	2
10Gb Ethernet	0	0	1.25

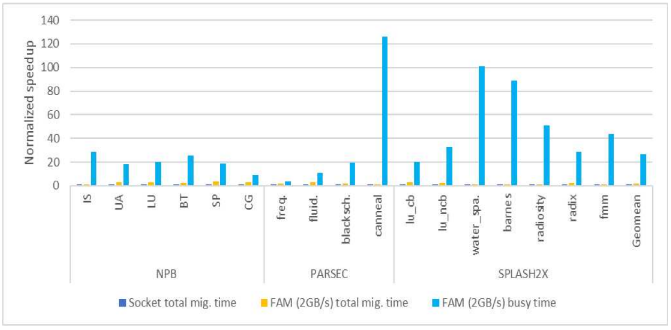
SPLASH-2X [24] native input benchmark suites. All workloads are migrated after executing twenty seconds, with the exception of "NPB IS" which migrated after five seconds for one thread and two seconds for four threads, and "SPLASH-2X radix" is migrated after five seconds for four threads.

We further examine the migration of REDIS [25] to investigate application performance degradation during live migration. REDIS is an in-memory data structure store and can be employed as database or in-memory cache. Through loading YCSB (Yahoo! Cloud Serving Benchmark) [26] records into REDIS, migrating REDIS to the target, and immediately accessing REDIS with YCSB at the target before the migration is finished, we can observe the impact of page fault overhead on REDIS performance with different YCSB workloads.

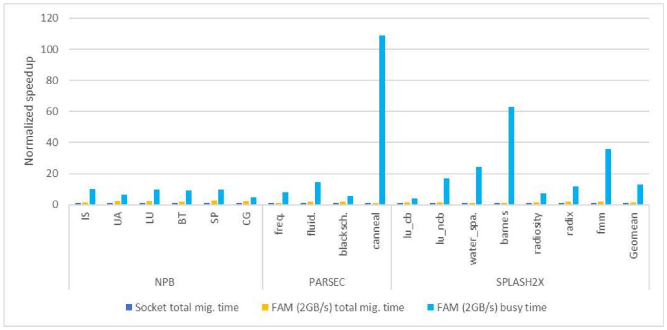
To evaluate FAM-aware migration, two limitations must be overcome. First, CRIU requires migrated applications to use the original pid they were checkpointed with. This means application cannot be "lively" migrated within the same physical host. Next, we do not have real FAM hardware, so we do not have a global, shared NVM across physical nodes. These two limitations seem to contradict with each other. Fortunately, with the help of a container virtualization technique, FAM can be emulated within a host machine by means of container and NVM: Two docker containers [27] are treated as target and source nodes; the emulated NVM, which is bind-mounted into containers so applications in containers can access NVM concurrently, can be emulated as FAM in our platform.

To compare the performance of our FAM-aware with existing CRIU post-copy, we assume all process states, except for memory pages, are migrated through FAM, so post-copy migration contains only 2 steps: checkpoint from source container and restart at target container. However, the methods of page transfer are different: one is by FAM, the other is through socket interface.

Furthermore, since NVM (FAM) is emulated from DRAM,



(a) One thread per benchmark. (Higher is better.)



(b) Four threads per benchmark. (Higher is better.)

Fig. 6: The busy time and total migration time performance of FAM-aware and existing CRIU post-copy migration using FAM. Expiration time is 100ms.

to correctly emulate the "slow" NVM, we use the same method proposed by Volos et al. [6] to add extra delay (via busy waiting) when accessing slow FAM (or Ethernet). Fig. 5 illustrates our delay model. The delay is added both when writing to and reading from FAM. The latency resulted from Gen-Z fabric is negligible compared to that of PCM [28]. Therefore, we only consider the access latency and bandwidth of NVM (PCM or NVDIMM). The delay calculation formula is as follows:

$$Delay_{R/W} = Latency_{R/W} + (data/bandwidth)$$

The parameters of NVM (and 10 Gigabit Ethernet used later) are shown at Table I. The bandwidth of NVM (PCM) is assumed to be 2GB/s [29]. Therefore, to access a single 4KB page from FAM, we have to wait around 3us and then can access FAM.

B. Evaluation Results

Fig. 6 (a) and (b) show the normalized total migration time and busy time performance of our FAM-aware vs. existing[2] CRIU post-copy with one and four threads. All the measurements are normalized to the total migration time of the existing post-copy (socket) of the same benchmark. FAM and socket (in Fig. 6) stand for the mechanism of page transfer by

[2]Note: for existing CRIU post-copy, the busy time equals to the migration time, so only total migration times are shown.

TABLE II: Average improvements of FAM-aware post-copy vs. existing CRIU post-copy.

# of threads	Mig. time imprvmt	Busy time imprvmt
1	2.00X	26.74X
4	1.63X	12.72X

FAM and by socket (network). We will keep using these terms hereafter. FAM (2GB/s) means the extra delay is added based on the 2GB/s bandwidth of PCM. The expiration time of active pushing timer is set as 100ms. Remember this timer would be reset whenever a page fault happens. We think 100ms should be a reasonable duration for workloads to access all working set pages before timer expires. So, the total migration time here should be a good indicator toward the different page fault overhead of these two mechanisms.

Tab. II summarizes the performance improvements of our FAM-aware post-copy. The number is the geometric mean of all benchmarks. From those results, we could conclude some useful observations.

First, instead of workload behavior, the busy time (also checkpoint time) of post-copy is impacted mostly by the dumped memory size. Thus, they are relatively small compared to total migration time.

Second, the improvements of total migration time (2X and 1.63X) come from faster demand paging handling, proving that our FAM-aware migration incurs less page fault overhead.

Third, the improvements of both total migration time and busy time reduce as the number of threads increases. For migration time, that is mainly because of the available bandwidth. We limit the FAM bandwidth as 2MB/s (PCM). On the other hand, although the existing CRIU acquires pages from source through socket, we do not apply any bandwidth limitation on it. Docker containers, in a single host, utilize Linux bridge component and bypass the NIC (network interface card) to communicate with each other. To estimate the bandwidth between docker containers, we use iperf3 [30] tool to measure and get the average 7.0 GB/s throughput (compared to 2GB/s at FAM-aware case). So, we could conclude that the reduction of improvements of total migration time when more threads are executed comes from the bandwidth limitations of FAM.

For busy time, the busy time of FAM-aware post-copy is the checkpoint time, which is almost the same if the memory footprint does not change, regardless of the number of executing threads. As to the existing post-copy, since the busy time is the total migration time, which would be improved as the number of threads increases because more threads will access pages more quickly. Therefore, the decreasing the busy time improvement results from the total migration time improvement of existing post-copy as more threads are executed.

Fig. 7 shows a similar comparison of FAM-aware and existing CRIU post-copy except that the expiration time is set as 0ms. Only the workloads whose migrated memory sizes are larger than 1GB are selected. All workloads are executed

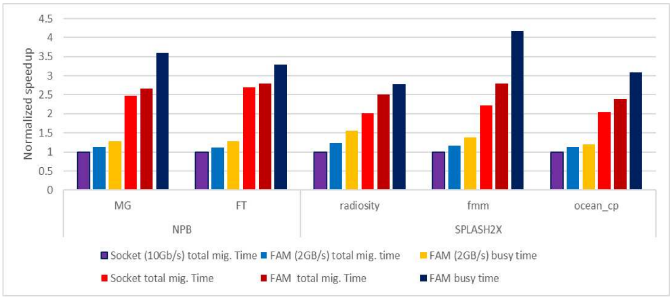


Fig. 7: The comparison of busy time and total migration time performance with the expiration time is 0ms. (Higher is better.)

TABLE III: Average improvements of FAM-aware post-copy for realistic case (FAM (2GB/s) v.s. socket (10Gb/s)) and ideal case (FAM v.s. socket).

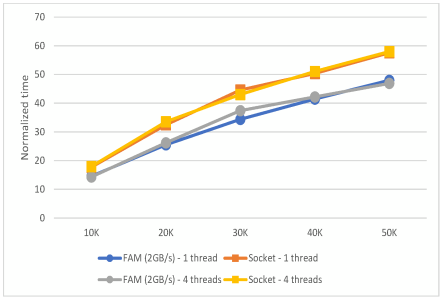
	Mig. time	Busy time
FAM (2GB/s) v.s. socket (10Gb/s)	15.44%	33.68%
FAM v.s. socket	15.16%	47.08%

with one thread and are migrated after executing twenty seconds. All the measurements are normalized to the total migration time of socket (10Gb/s) of the same benchmarks. Socket (10Gb/s) is assumed that the employed underlying network is 10 Gigabit Ethernet. FAM (without bandwidth limitation) means no delay is added when accessing NVM, which can be treated as the case of NVDIMM and whose performance is also the best among all cases. In our platform, the average throughput of accessing DRAM via file system `write` is around 6.6 GB/s (a little lower than 7 GB/s of socket throughput between containers).

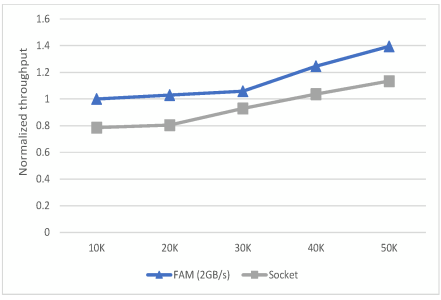
The 0ms expiration time means that the lazy-page daemon would actively push the pages from source from the beginning. Meanwhile, if a page fault happens, the daemon will also try to handle page fault event at best effort. From Fig. 7, we could conclude that (A) the bandwidth provided by the transmission media dominates the migration performance; (B) if the bandwidth is close to each other, FAM-aware is better than existing post-copy due to the lighter overhead. Tab. III summarizes the improvements.

Finally, REDIS and YCSB are utilized to investigate the migration performance, especially for performance degradation. 500K records are loaded by YCSB into REDIS at the source node, and each record is of 100 fields and the size of each field is fixed to 10B. This configuration will result in 939MB of pages to be migrated. After downtime, YCSB accesses REDIS at target immediately by different operation records (from 10K to 50K). The YCSB workloads are all configured as uniform distribution, readallfields and writeallfields are false, and R/W ratio are 70/30. The YCSB employs one and four threads to access REDIS. Fig. 8 shows the results. (a) to (c) employ 150ms expiration time and (d) to (f) employ 3ms.

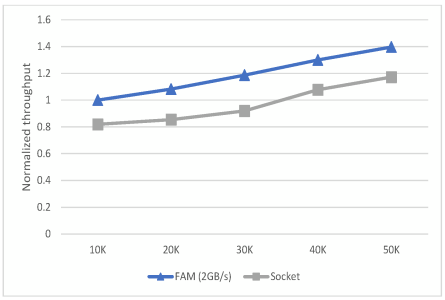
Fig. 8 (a) shows the measured total migration times normalized to the busy time of FAM (2GB/s). The busy times of one



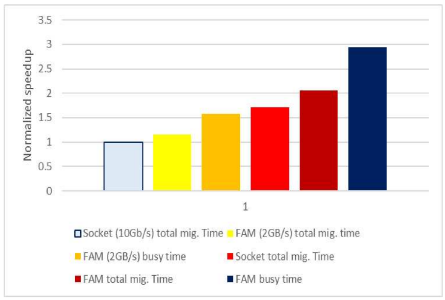
(a) Normalized total migration time. (Lower is better.)



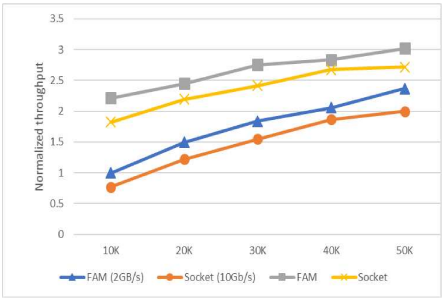
(b) Normalized throughput with one thread. (Higher is better.)



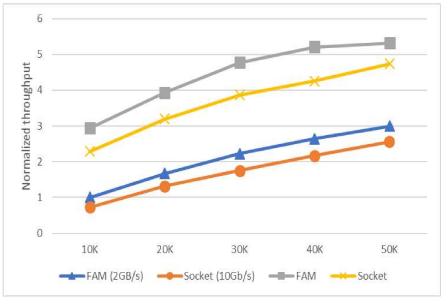
(c) Normalized throughput with four threads. (Higher is better.)



(d) Normalized total migration time and busy time performance. (Higher is better.)



(e) Normalized throughput with one thread. (Higher is better.)



(f) Normalized throughput with four threads. (Higher is better.)

Fig. 8: Migration performance of REDIS accessed by YCSB. (a) to (c): The expiration time is 150ms; (d) to (f): The expiration time is 3ms.

TABLE IV: Average REDIS throughput improvements of FAM-aware post-copy of real case (FAM (2GB/s) v.s. socket (10Gb/s)) and ideal case (FAM v.s. socket).

	1 thread	4 threads
FAM (2GB/s) v.s. socket (10Gb/s)	19.8%	25.69%
FAM v.s. socket	12.7%	21.79%

and four threads are almost the same since the same amount of data (939MB) are checkpointed. FAM-aware post-copy improves average total migration time 23.92% and 22.48% with one and four threads respectively. Fig. 8 (a) also indicates that the total migration time is not related to the number of threads of YCSB. The reason is that the REDIS is single-threaded, so more requests (threads) from YCSB cannot make the pages of REDIS be accessed faster. Fig. 8 (b) and (c) show the REDIS performance during migration. FAM-aware post-copy lets the REDIS perform 22.3% and 23.4% higher with one and four threads. respectively. Fig. 8 (a), (b), and (c) could prove that the total migration time and application degradation have some correlations because they are impacted mostly by the latency of page fault handling if the expiration time is larger enough.

Fig. 8 (d) shows the normalized total migration time and busy time performance with 3ms expiration time. Because those times at different operation counts of YCSB are almost the same, we only take the average. This result also looks like Fig. 7. Tab. IV summarizes the results of Fig. 8 (e) and (f).

Again, Fig. 8 (d), (e), and (f) also show that the total migration time and performance degradation are both influenced by the bandwidth of transmission media if the expiration time is too small and active pushing is triggered soon enough. The throughput of REDIS increases as the number of operations increases; this is because the chances of page fault reduce as more pages are migrated.

VI. Conclusion

We presented FAM-aware, checkpoint-based, post-copy migration. Through FAM, a global, shared NVM pool in a rack scale, we can map the entire migrated memory space onto FAM. So, the data migration can be simplified as memory-semantics to achieve a much lower page fault latency path. We have implemented our prototype at CRIU, and shown that our approach has lower busy time (at least 33%), lower total migration time (at least 15%), and migrated application can perform at least 12% better (i.e. lower application degradation) during migration process.

Acknowledgments

We first would like to thank Dr. Jaemin Jung, who was a postdoc at Texas A&M University, for his helpful comments and suggestions about this paper. We also thank the anonymous reviewers for their valuable and useful comments and feedback to improve the content and quality of this paper. Finally, we want to thank the National Science Foundation, which supports this work through grants I/UCRC-1439722 and

FoMR-1823403, and generous support from Hewlett Packard Enterprise.

REFERENCES

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, pp. 131–141, Mar. 2010.

[2] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS '12*. London, England, UK: ACM, Mar. 2012, pp. 401–410.

[3] "Intel optane technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[4] Gen-Z speciticaions. https://genzconsortium.org/specifications/.

[5] K. Keeton, "Memory-driven computing," in *FAST '17*. Santa Clara, CA: USENIX, 2017.

[6] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS '11*. Newport Beach, CA: ACM, Mar. 2011, pp. 91–104.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS '11*. Newport Beach, CA: ACM, Mar. 2011, pp. 105 – 118.

[8] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *FAST '15*. Santa Clara, CA: USENIX, Feb. 2015, pp. 167 – 181.

[9] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Speculative paging for future nvm storage," in *MEMSYS '17*. Alexandria, Virginia: ACM, Oct. 2017, pp. 399 – 410.

[10] L. Liang, R. Chen, H. Chen, Y. Xia, K. Park, B. Zang, and H. Guan, "A case for virtualizing persistent memory," in *SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing*. Santa Clara, CA: ACM, Oct. 2016, pp. 126 – 140.

[11] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *ASPLOS '15*. Istanbul, Turkey: ACM, Mar. 2015, pp. 3 – 18.

[12] K. Ye, D. Huang, X. Jiang, H. Chen, and S. Wu, "Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective," in *GREENCOM-CPSCOM '10*. IEEE, Dec. 2010, pp. 171–178.

[13] K. Chanchio and X.-H. Sun, "Communication state transfer for the mobility of concurrent heterogeneous computing," *IEEE Transactions on Computers*, vol. 53, pp. 1260–1273, 2004.

[14] E. R. Zayas, "Attacking the process migration bottleneck," in *SOSP '87*. Austin, TX: ACM, Nov. 1987, pp. 13–24.

[15] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 14–26, Jul. 2009.

[16] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *CCEM '12*. Bangalore, India: IEEE, Oct. 2012.

[17] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang, "Live migration of multiple virtual machines with resource reservation in cloud computing environments," in *CLOUD '11*. Washington, DC: IEEE, Jul. 2011.

[18] M. F. Bari, M. F. Zhani, Q. Zhang, R. Ahmed, and R. Boutaba, "CQNCR: Optimal VM migration planning in cloud data centers," in *IFIP '14*. Trondheim, Norway: IEEE, Jun. 2014.

[19] Checkpoint/Restore In Userspace (CRIU). https://www.criu.org/.

[20] C. Jo, E. Gustafsson, J. Son, and B. Egger, "Efficient live migration of virtual machines using shared storage," in *VEE '13*. Houston, TX: ACM, Mar. 2013, pp. 41–50.

[21] Emulate NVDIMM in Linux. https://nvdimm.wiki.kernel.org/.

[22] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks-summary and preliminary results," in *SC '91*. ACM, 1991, pp. 158–165.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT '08*. Toronto, Ontario, Canada: ACM, Oct. 2008, pp. 72–81.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA '95*. S. Margherita Ligure, Italy: ACM, Jun. 1995, pp. 24–36.

[25] Redis: An in-memory data structure store. http://redis.io/.

[26] B. F. Cooper, A. Silberstein, ErwinTam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC '10*. Indianapolis, Indiana: ACM, Jun. 2010, pp. 143–154.

[27] Docker container. https://www.docker.com/.

[28] "Gen-z overview," https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview.pdf.

[29] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *IPDPS '13*. Boston, MA: IEEE, May 2013.

[30] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/iperf-download.php.