Optimized Pre-Copy Live Migration for Memory Intensive Applications

Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, Eric Roman
Lawrence Berkeley National Laboratory
Email: {kzibrahim, shofmeyr, cciancu, eroman}@lbl.gov

***Abstract*—Live migration is a widely used technique for resource consolidation and fault tolerance. KVM and Xen use iterative pre-copy approaches which work well in practice for commercial applications. In this paper, we study pre-copy live migration of MPI and OpenMP scientific applications running on KVM and present a detailed performance analysis of the migration process. We show that due to a high rate of memory changes, the current KVM rate control and target downtime heuristics do not cope well with HPC applications: statically choosing rate limits and downtimes is infeasible and current mechanisms sometimes provide sub-optimal performance. We present a novel on-line algorithm able to provide minimal downtime and minimal impact on end-to-end application performance. At the core of this algorithm is controlling migration based on the application memory rate of change.**

I. Introduction

Virtualization technologies are ubiquitously deployed in data centers and offer the benefit of performance and fault isolation, flexible migration [16] , resource consolidation [13], and easy creation [5] of specialized environments. They have been extensively used to run web server, E-commerce and data mining workloads. Recently, these workloads have been supplemented with High Performance Computing (HPC) applications: the Amazon Elastic Compute Cloud (EC2) already provides virtualized clusters targeting the automotive, pharmaceutical, financial and life sciences domains. The US Department of Energy is evaluating virtualization in the Magellan [12] project. Virtualization is also evaluated in very large scale HPC computing systems such as the Cray XT4, where it has been shown [11], [10] to scale up to 4096 nodes.

Live virtual machine migration is one of the key enabling technologies for fault tolerance and load balancing. With live migration, whole environments comprised of Virtual Machine (VM), Operating System and running tasks, are moved without halting between distinct physical nodes or Virtual Machine Monitors (VMM). Live migration has been extensively studied for commercial workloads where tasks are often independent and serve short lived requests; in contrast, HPC workloads have tasks tightly coupled by data movement and tend to persistently use a significant fraction of the system memory. Due to these differences, current live migration techniques do not cope well with HPC workloads.

In this study we examine OpenMP and MPI applications and propose a novel control technique specifically designed for live migration of scientific applications. We implement our algorithms in KVM [9], which uses a state-of-the-art *iterative pre-copy* mechanism. The contributions of this paper are summarized as follows:

- We extend previous studies of migration performance [7], [15] to quantify the impact of: i) the monitoring mechanism for dirty pages; ii) the memory and the network contention; and iii) the parallel programming model and the application data set.
- We present a novel KVM pre-copy algorithm with different termination criteria that it is able to achieve a minimal downtime and minimizes the impact on the application end-to-end performance. Our algorithm can be trivially ported to other implementations, such as Xen.

In KVM, the performance of iterative pre-copy migration is controlled by the bandwidth allocated, as well as a target downtime for the last iteration. These two metrics are chosen by system administrators to both minimize the migration overhead, and satisfy service level agreements (SLAs) that require guarantees on the maximum downtime. For HPC applications, downtimes are generally not as important as overall execution time, although downtime can result in application failure due to factors such as network timeouts. Compared to commercial server applications, HPC applications tend to be memory intensive, and as our analysis in Sections IV and V shows, memory intensive applications are difficult to migrate because rate limits and downtime cannot be optimally set without detailed knowledge of the application behavior. In particular, state-of-the-art pre-copy techniques as implemented in KVM 0.14.0 do not converge (iterate indefinitely) for some of our benchmarks when executed with multiple processors per VM.

Previous work on pre-copy live migration for scientific applications [7], [15] reports acceptable performance on systems with one or two cores per node. In the rest of this paper we argue that when increasing the number of cores or memory per node, controlling pre-copy live migration using rate limits and target downtimes can become infeasible and propose a an approach that takes into account only the application's or VM's rate of memory changes. This is particularly suitable for memory-intensive HPC applications where we do not need to guarantee limits on downtimes.

In Section VII we present a novel *iterative pre-copy* algorithm designed for convergence instead of guaranteeing maximum downtime. In our experiments, the algorithm always converges within the first third of the application execution, significantly reducing the end-to-end application performance

impact (in some cases by an order of magnitude over the default KVM settings). At the core of this algorithm is making migration decisions based solely on detecting memory activity patterns and switching to stop-and-copy migration when further reductions in downtime are unlikely to occur. In practise, our approach also provides low downtime and can be easily retrofitted into implementations that need to provide SLAs.

II. Virtual Machine Migration

In live migration, the virtual machine (or the application) keeps running while transferring its state to the destination. A helper thread iteratively copies the state needed while both end-points keep evolving. The number of iterations determines the duration of live migration. As a last step, a stop-and-copy approach is used, its duration is referred to as *downtime*. All implementations of live migration use heuristics to determine when to switch from iterating to stop-and-copy. Throughout this paper we refer to this decision point as *convergence* or termination.

Iterative Page Pre-Migration *(pre-copy)***:** This technique starts by copying the whole source VM state to the destination system. While copying, the source system remains responsive and keeps progressing all running applications. As memory pages may get updated on the source system, even after they have been copied to the destination system, the approach employs mechanisms to monitor page updates. Pre-copying is implemented in KVM and Xen. Previous studies [4], [7] indicate that the performance of the approach is determined by the network bandwidth available to the migration process.

The performance of live VM migration is usually defined in terms of migration time and system downtime. All existing techniques control migration time by limiting the rate of memory transfers while system downtime is determined by how much state has been transferred during the "live" process. Minimizing both of these metrics is correlated with optimal VM migration performance and in KVM it is achieved using open loop control techniques.

With open loop control, the VM administrator sets configuration parameters for the migration service thread, hoping that these conditions can be met. The input parameters are a limit to the network bandwidth allowed to the migration thread and the acceptable downtime for the last iteration of the migration. Setting a bandwidth limit while ignoring page modification rates can result in a backlog of pages to migrate and prolong migration. Setting a high bandwidth limit can affect the performance of running applications. Checking the estimated downtime to transfer the backlogged pages against the desired downtime can keep the algorithm iterating indefinitely. Approaches that impose limits on the number of iterations[1] or statically increasing the allowed downtime can render live migration equivalent to pure stop-and-copy migration.

[1]As discussed in Section IX, Xen defaults to this approach.

A. The KVM Implementation

KVM, as of the QEMU-KVM release 0.14.0, uses Algorithm 1. We show the pseudo-code for the call-back function invoked by the helper thread that manages migration. This code runs sequentially, regardless of the number of processors in the virtual machine and it proceeds in three phases.

In the first phase (lines 2-10), all memory pages are marked dirty and the modification tracking mechanism is initialized. In the second phase, pages are copied if they are marked dirty. The dirty bit is reset, but can be set again if the application has modified the page. Normally, this second phase is the longest in iterative pre-copy Pages are being copied as long as the maximum transfer rate to the destination machine is not exceeded. Pages that are modified but not copied are used to estimate the downtime if the migration proceeds to the third stage. If the estimated downtime is still high, the algorithm iterates until it detects a value lower than the target value. When the target downtime is met, the migration enters the third and final stage, where the source virtual machine (and applications) is stopped. Dirty pages are transmitted to the destination machine, where execution is resumed.

B. Live Migration Optimization Challenges

Figure 1 illustrates the challenges faced when trying to optimize migration while maintaining application performance. The results are obtained for the OpenMP implementation of NAS MG, running on two cores. As OpenMP does not use the network, the experiment is performed with no migration rate limit, *i.e.*, the best achievable migration speed.

The left hand side shows the variation of downtime with migration time. Typically, the downtime decreases as we increase the migration time because the inactive datasets are transmitted to the destination before stopping the machine for the last migration. As the figure illustrates, choosing a downtime that is neither excessive or unattainable can be difficult even for this simple case. Furthermore, as shown later, this typical curve varies greatly depending on the workload characteristics and the VM configurations.

The right hand side of Figure 1 shows the variation of the performance of the application running on the VM as a function of migration time. Increasing the migration time increases application execution time and consequently, the migration time should be minimized for optimal performance.

Several problems are apparent when examining the two plots: i) a static target downtime can be easily requested below the attainable downtime, resulting in increased migration time or failure to migrate the virtual machine; ii) best application performance is obtained with a short migration time, which in turn determines an increased downtime.

In the rest of this paper we argue that when increasing the number of cores or memory per node, controlling pre-copy live migration using rate limits and target downtimes can become infeasible and propose a novel approach that takes into account only the application's or VM's rate of memory changes.

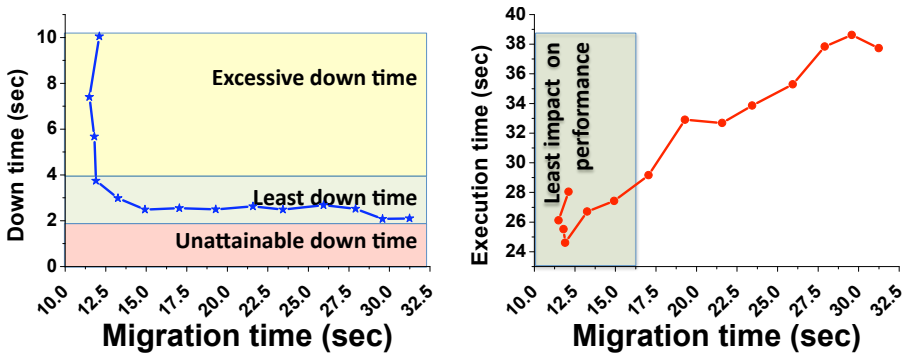

Fig. 1. *On the left, a typical VM migration plot for the downtime* vs. *the duration of live migration. The impact on performance by increasing the duration of live migration is shown on the right. While downtime decreases when live migration is prolonged, the execution time is negatively impacted.*

Algorithm 1 Pseudo code for the main call-back for live VM migration as implemented in QEMU-KVM.

```
1:  procedure RAM_SAVE_LIVE(sid, stage)
2:      if stage = 1 then
3:          set_dirty_tracking()
4:          for all block_i in ram_list do
5:              for all page_j in a block_i do
6:                  set the migration dirty bit for page_j.
7:              end for
8:          end for
9:          next_stage ← 2
10:     end if
11:     start_time ← get_time()
12:     bytes_transferred ← 0
13:     while transfer_rate(sid) < max_rate do
14:         bytes_sent ← migrate_page(sid)
15:         add bytes_sent to bytes_transferred
16:         if bytes_sent = 0 then
17:             break                          ▷ exit while loop
18:         end if
19:     end while
20:     transfer_time ← get_time() − start_time
21:     bandwidth ← bytes_transferred/transfer_time
22:     if stage = 3 then
23:         while migrate_page(sid) ≠ 0 do
24:         end while
25:         reset_dirty_tracking()
26:     end if
27:     remaining ← remaining_modified_data()
28:     expected_downtime ← remaining/bandwidth
29:     if expected_downtime < max_downtime then
30:         next_stage ← 3
31:     end if
32:     return next_stage
33: end procedure
```

III. Experimental Setup

We experiment with the KVM open source virtualization technology, version 0.14.0 and the Linux kernel 2.6.32.8. In our experiments, each virtual machine is configured with 1GB of memory per core, e.g. a 4 core VM used 4GB of memory. The system used for the evaluation contains two 1.6 GHz quad-core quad-socket UMA Intel Xeon E7310 (Tigerton) nodes connected with dual InfiniBand and Ethernet networks. The achievable bandwidth is 800 MB/s and 120MB/s over InfiniBand and Ethernet respectively. In our environment we use the *virtio* [18] driver which provides near native Ethernet performance in KVM. KVM on Tigerton uses the state-of-the-art Intel VT-x support for hardware virtualization. Each experiment has been repeated six times and the average is reported when not mentioned otherwise.

As a workload we use implementations of the NAS Parallel Benchmarks [2] in two popular parallel programming paradigms: MPI (OpenMPI 1.4.2 with `gcc` 4.3.2) and OpenMP (`gcc` 4.3.2 with GOMP). We perform experiments with up to eight cores per VM for the problem classes A, B and C and, overall the memory footprint of the workload varies from tens of MBs to tens of GBs. Asanović et al [1] examined six different promising domains for commercial parallel applications and report that a surprisingly large fraction of them use methods encountered in the scientific domain. In particular, all methods used in the NAS benchmarks appear in at least one commercial domain. Thus, beside their HPC relevance, these benchmarks are of interest to other communities.

To understand the performance aspects of live migration we perform two sets of experiments: 1) the application and the migration run concurrently over Ethernet and; 2) the application runs over Ethernet while the migration uses the InfiniBand network. The former is presented in Sections IV and V and captures the *worst case* scenario, which occurs in bandwidth starved environments or environments where the memory rate of change is much higher than the network draining rate. The latter is presented in Section VI and captures the *best case* scenario, where high bandwidth is available for migration and there is the least degree of interference between the application and the migration process. Furthermore, by running the application over a slower network we also lower

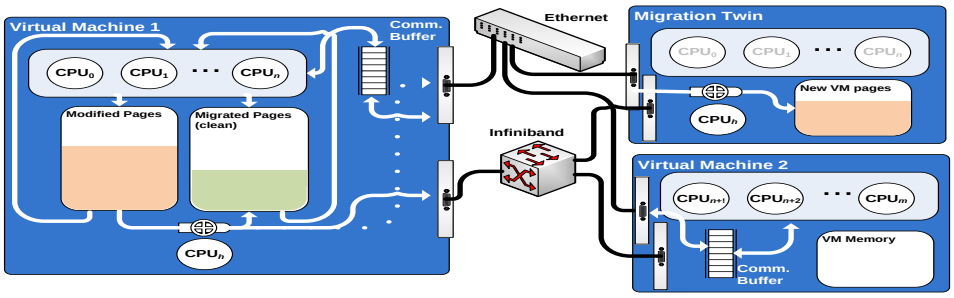

Fig. 2. *The virtual machine (VM) configuration used in the experiments throughout the paper. Depending on the workload type (OpenMP vs. MPI) one or two VMs, respectively, are used in addition to the migration destination.*

its rate of memory changes, as MPI ranks will wait longer for communication with other off-node ranks.

IV. Impact of Migration on Application Performance

Several factors determine the impact of migration on end-to-end application performance:

- Monitoring page modifications causes additional TLB flushes and soft page faults on write operations.
- Copying modified pages to the IO buffer used for the transmission to the destination machine causes memory contention.
- Running the application and the migration process concurrently causes network contention.
- Only one service thread per VM handles migration, but the application can use multiple cores.

We quantify these overheads by incrementally modifying the migration mechanism. To measure the impact of dirty page tracking, we modified the helper thread to perform only monitoring without sending data to the destination virtual machine. To add the effect of memory contention, we enable copying of modified pages to the communication buffers, without actual data transmissions. To quantify the total impact on performance, we start the migration process and allow it to proceed until the completion of the application.

Figures 3 and 4 show the cumulative impact on the OpenMP and MPI implementations of NAS classes A and B, respectively. All experiments are performed using the Ethernet network and we present configurations using one, two and four cores per virtual machine. The figures show that, for both OpenMP and MPI, the additional page faults impact application performance considerably. For OpenMP and MPI we observe an average slowdown of 20% for configurations with one core per virtual machine. As we increase the number of cores per virtual machine, the overhead of the monitoring mechanism increases. Using four cores per virtual machine, we observe average slowdowns of 55% and 65% for OpenMP and MPI respectively, For both OpenMP and MPI, the impact of the memory contention is modest and it accounts for a few percent of the total slowdown.

These results suggest that we might improve live migration performance by allowing more parallelism in the monitoring process, i.e. allowing multiple migration threads to execute concurrently. Due to space constraints we do not explore this further in the paper.

The MPI experiments confirm that the dominating factor is the available network bandwidth. Increasing the number of cores per VM degrades the application performance, so that we observe average slowdowns of 130%, 190% and 590% for VMs using one, two and four cores respectively. Since migration is allowed to proceed until the application terminates, these overheads reflect either upper bounds in the case when migration terminates earlier, or actual impact when the migration process does not converge. As we will show, the latter is a common occurrence in practice for this workload. For these MPI experiments we do not perform any rate limitation. Adding rate limitation mechanisms is likely to increase the duration of the migration, which in turn increases the impact on end-to-end application performance as indicated by Figure 1.

V. Impact of Applications on Migration Performance

Two application-specific factors that affect the performance of the migration process are:

1) the rate at which the application changes memory, and
2) the programming model. An OpenMP application is contained within one VM, but MPI applications can span multiple VMs.

All experiments presented in this section are performed using the Ethernet network. In Figure 5, we show the evolution of downtime for the OpenMP implementations when we control the number of iterations before changing from live to stop-and-copy migration. Each iteration is set to be at most 1000 ms. For reference, the default allowed downtime in qemu-KVM is 30 ms, configurable by system administrators.

As illustrated, for small datasets (class A) running on single core VMs, the downtime converges rapidly below 30 ms for most applications (except LU and SP). In these cases the default KVM configuration will succeed in attaining the target downtime. Increasing the dataset size to class B and running two cores per VM, notice that most applications cannot achieve the default downtime of 30 ms and FT requires as much as seven seconds. Increasing the VM concurrency to four for the same dataset (class B) worsens the downtime and we observe durations anywhere between two and 30 seconds. The slope of reducing the downtime is also worse

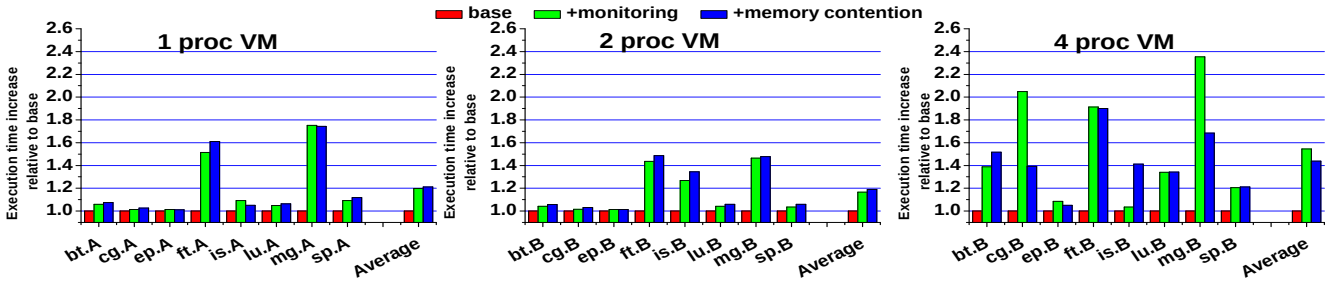

Fig. 3. *Impact of VM migration on the end-to-end performance of the NPB3.3 OpenMP benchmarks, classes A and B.*

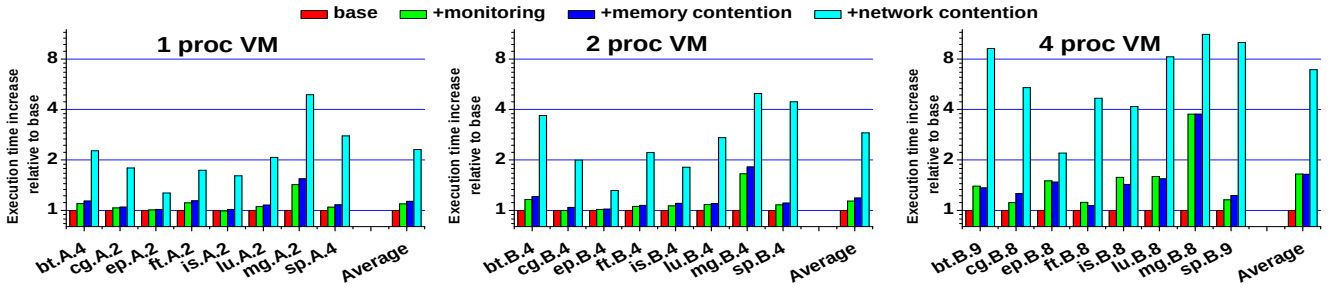

Fig. 4. *Impact of VM migration on the end-to-end performance of the NPB3.3 MPI benchmarks, classes A and B.*

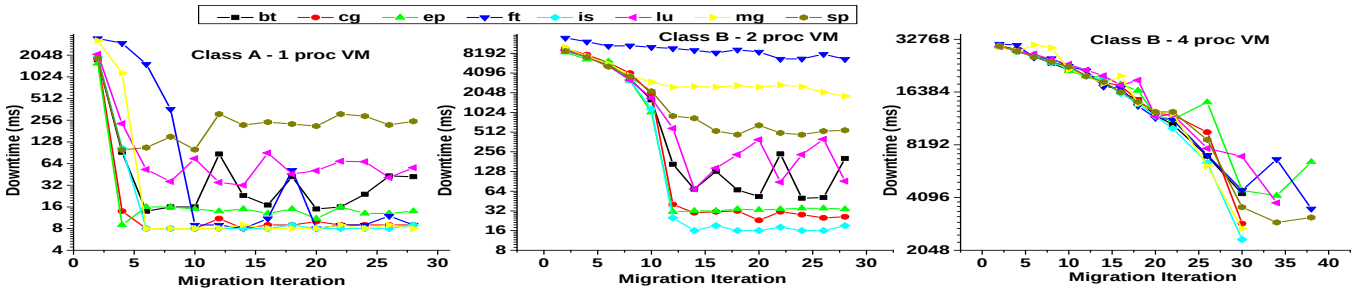

Fig. 5. *Downtime after each migration iteration for different VM configurations running NPB3.3 OpenMP class A and B. Each iteration is at most 1000ms.*

in this case: the least downtime is achieved after a much larger number of iterations. These results show the difficulty of choosing a system wide target downtime and also indicate that heuristics that impose static limits on the number of iterations are likely to produce sub-optimal downtimes when increasing the number of cores per VM.

We repeated the same experiments for the MPI benchmarks with similar results. For MPI applications, the additional network contention determines downtimes in the seconds range, even for simple configurations with small data sets and few cores per VM. The MPI case poses additional challenges as large downtimes and low responsiveness cause in our experiments application failures due to timeouts in the MPI layer. Maintaining responsiveness of the machine requires reducing the flow of migration which in turn increases the downtime. Thus, besides the difficulty of choosing the rate limit and the target downtime, the MPI applications require a feedback loop with the MPI library timeouts and downtime is transformed from a mere performance metric to a factor determining application failure.

The downtime evolution is explained when examining the relationship between the application working set (writable working set, Clark et al [4]) and the network bandwidth. Figure 6 illustrates the two possible scenarios. In the first case the network capacity is higher than the workload capability of writing new pages and the downtime decreases with a high slope. In the second case, the memory rate of change is higher than the network capability and the number of migrated pages is always less than the number of modified pages: downtime remains constant or decreases with a low slope.

Figure 7 shows the active dataset and the corresponding network performance required for draining it at different processor counts. We instrument the monitoring activity and measure the number of modified data pages using a sampling interval of 1000ms. We reset the pages marked dirty and compute the bandwidth needed to handle their transfer.

We consider the effective range as those in the interquartile range (25% to 75%), *i.e.* those within the box in the figure. Figure 7 illustrates two important points. First, the working dataset size varies greatly among applications. The largest size is associated with FT, which explains its large required downtime in Figure 5. Second, the network performance required to drain the active working set is high, up to 1.5GB/s.

For the MPI case we monitor the activity in only one of the virtual machines allocated to the application. The active working set is generally larger for the MPI applications and the bandwidth requirement is higher than in the OpenMP case. Increasing the parallelization of the OpenMP applica-

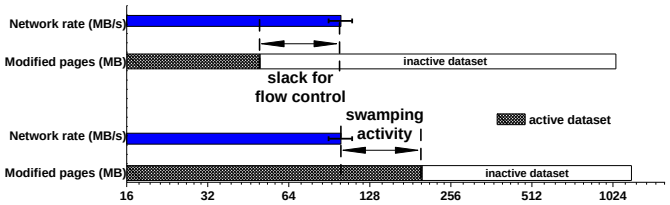

Fig. 6. *Two possible scenarios of application active dataset* vs. *migration network activity. Rate limiting can be effective if a slack exists in networking performance compared with the size of the active set.*

tions increases the required bandwidth, while for the MPI applications our modified page monitoring reaches the point of impairing the generation of modified pages when we increase parallelization.

Overall, these results indicate that applications running inside parallel virtual machines at the available core concurrency easily overwhelm, even with small datasets, the available network draining rate. They also indicate that rate limitation and target downtimes are infeasible metrics to control the migration process. As these results were obtained for the worst case scenario (low bandwidth over Ethernet) a valid question is whether these trends are observable when high network bandwidth is available for migration. As shown next, even for the best case scenario with independent migration over InfiniBand/RDMA, they persist when increasing the dataset size or the core concurrency.

VI. Migration Using InfiniBand and RDMA

Since our experimental system has dual network cards, we further isolate the impact of live migration by separating the application traffic from the migration traffic and assess the impact under the *ideal scenario* of having a separate high speed interconnect for migration management. In the rest of the experiments, the applications are running over Ethernet and the migration proceeds over InfiniBand. In this setting, the migration has the least possible interference with the application execution.

We have implemented extensions to Qemu-KVM to support migration over InfiniBand using remote direct memory access (RDMA). For brevity we do not present further details but note that our implementation is as well optimized as the implementation of migration on Xen/InfiniBand presented by Huang et al [7], i.e. it overlaps page monitoring and transmission, it performs aggressive page coalescing and InfiniBand level flow control of outstanding transfers.

Currently, KVM does not support virtualization of InfiniBand devices, but it is only a matter of time before this support becomes available. Thus, we believe that besides providing the first open source implementation of KVM migration over InfiniBand, our design and evaluation provide valuable insights into tuning live migration over InfiniBand devices.

A. Performance with RDMA Migration

Figure 8 presents the evolution of the downtime with the increase in dataset size and number of active cores in the VM. As downtime is highly dependent on the particular time during execution at which migration is started, we present a statistical summary of downtime over the application lifetime: we report the downtime for a stop-and-copy after each iteration, for as many iterations required until application termination. The highest downtime usually occurs after the first iteration, while the lowest downtime occurs after the application termination. As shown, any increase in the dataset size or the number of processors determines an increase in downtime. The downtime ranges from tens of milliseconds to multiple seconds; MPI applications usually have a lower downtime than OpenMP applications. As expected, a large downtime is associated with application activity that exceeds the network capability for transferring the data.

Even with a separate network for migration, the impact on application performance is high. Figure 9 shows that the average slowdown due to the impact of the page monitoring mechanism is respectively 55% and 80% for the OpenMP and MPI workloads, on 4 core VMs. In the RDMA case, this impact is considerably higher than in the Ethernet case: the higher network bandwidth increases the degree of bandwidth contention to memory.

The average slowdown for MPI applications is 6.8x, much higher than the 1.8x average for OpenMP applications, even though the bandwidth used for migration is not at the application's expense. We explain the MPI behavior as follows. The MPI ranks synchronize with each other while migration implicitly slows down the application execution. While for OpenMP synchronization is contained within one VM, for the MPI case it spans multiple nodes: TCP/IP re-transmissions and remote node delays while waiting for synchronization both contribute to the slowdown. We did not separate these effects, but we conjecture that in general, applications spanning multiple nodes will observe a higher slowdown due to migration than single node applications.

While we observe a significant improvement of the overall performance in the case of high speed networks and no rate limits, the application's memory rate of change is still higher than the available draining rate and large downtimes and considerable application performance impacts are still present.

VII. Algorithm for Migration Convergence

Clearly, using a static target downtime as a condition for switching from iterative migration to stop-and-copy leads to sub-optimal behavior. We propose a new technique where switching is decided by matching memory update patterns that indicate that no beneficial progress is achieved by continuing live migration. As a progress measure we use the probability of further reduction in downtime, without requiring any static estimates.

The memory update patterns are summarized in Figure 10. The first pattern represents the case where the number of modified pages is not reduced by iterative pre-copying. In this case, the application's memory rate of change exceeds the available bandwidth and continuing migration will only

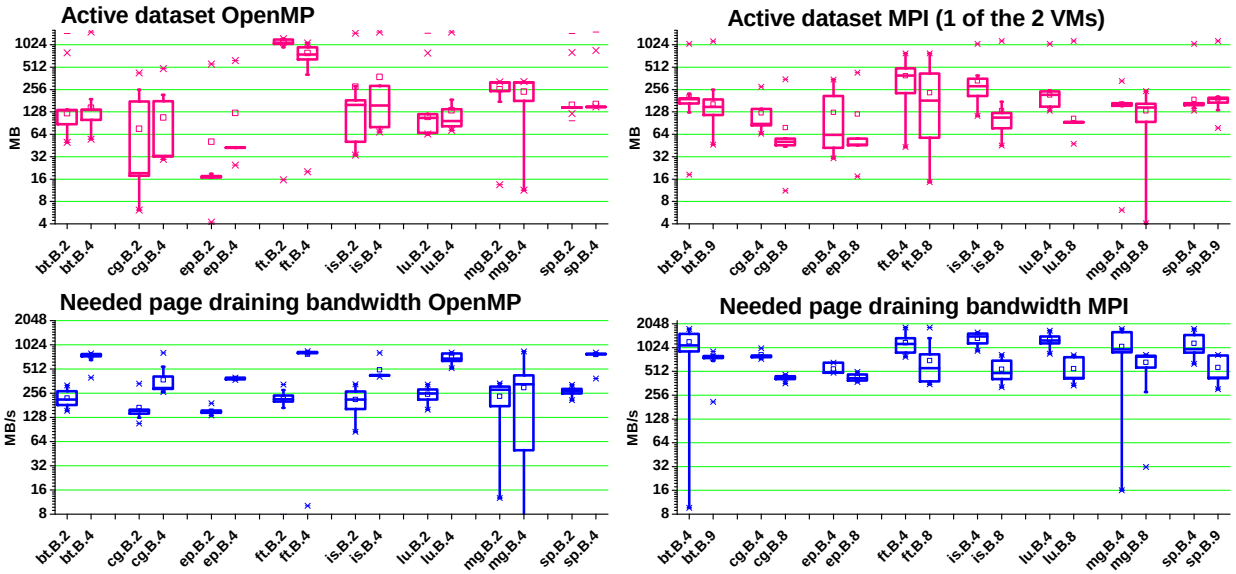

Fig. 7. *The application's active dataset sampled at 1000 ms. Data is summarized in quartile form (25% to 75%). Below: the network bandwidth required for transferring the active dataset within one sampling interval.*

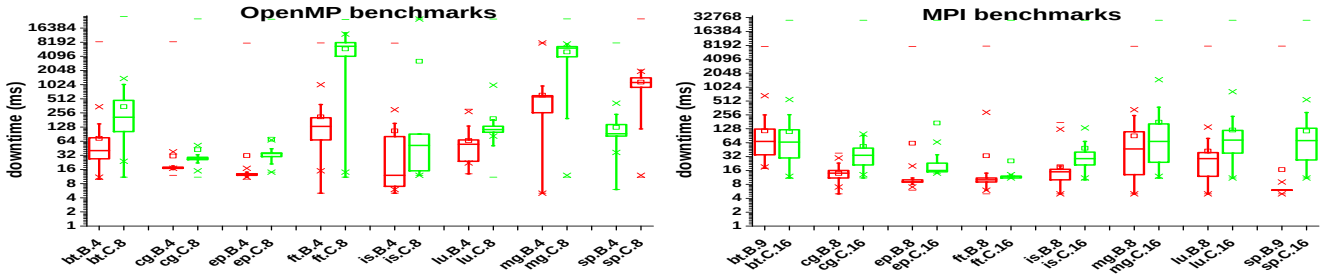

Fig. 8. *Downtime after each migration iteration for the optimized RDMA implementation. We present its variation with the VM configuration, programming model, and application dataset size (class of NPB benchmarks). Data is summarized in interquartile form.*

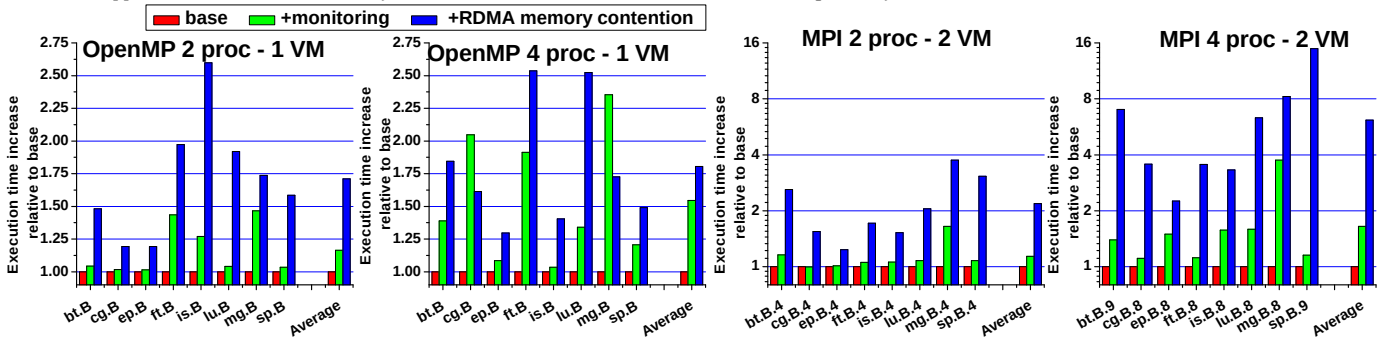

Fig. 9. *Impact of VM migration on the end-to-end performance of the NPB3.3 OpenMP and MPI benchmarks, class B. Migration is performed over InfiniBand.*

degrade end-to-end performance, without any downtime reduction. The second pattern is when the application's memory activity drops during execution such that a small downtime can be attained. In applications, this situation occurs for example in synchronization and barrier operations. The third pattern occurs when most of the transmitted pages are similar to those transmitted in the previous iteration: in this case the modified pages can be drained but the application performs an iterative computation on the dataset.

Algorithm 2 presents the details of our new approach. At the core of the implementation there is a mechanism to estimate the rate of page transmission during migration. This is essential for detecting the three different patterns shown in Figure 10.

Detecting pattern *a* requires monitoring the number of page changes per a constant time interval. As RDMA transfers are non-blocking (split into post and waiting for completion), we needed to know a priori the number of pages to post before checking the number of the remaining modified pages and then waiting for transfer completion. As shown in lines 25-37 of the algorithm, we use linear regression to estimate the number of pages to send in order to make the sampling interval constant.

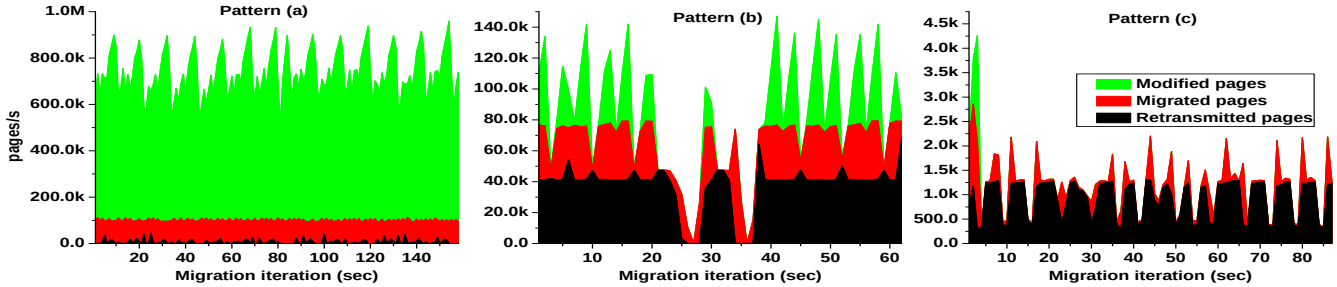

Fig. 10. *Three patterns to detect termination of the pre-copying mechanism: a) modified pages are stable for a prolonged period of time; b) application activity drops below a certain threshold; and c) the migration activity involves high retransmission of pages.*

In our experiments we use a sampling interval of one second. Shorter intervals increase the variation in the measurements of paging activity and make it more difficult to recognize steady state. Our implementation also computes a moving average in order to handle variation of the monitored data. We then use a simple filtering technique to check if the rate of page modification is stable. A stable region triggers a switch to the last stop-and-copy stage of migration.

For pattern *b* in Figure 10, our implementation detects when all modified pages have been transferred in an interval less than the preset sampling interval (omitted for brevity in Algorithm 2). In this case we know that we migrated all the inactive dataset and the active set is smaller than the interconnect capability for migration.

To track the page retransmission activity required for the detection of pattern *c*, the algorithm maintains a bitmap corresponding to the pages transmitted in the previous iteration. During any iteration, we count the number of retransmitted pages. If the percentage of retransmissions exceeds 90% of the migrated pages the algorithm proceeds to the last stage of migration (lines 41-43 in Algorithm 2). In our implementation, we maintain only one history bitmap for the previous iteration. The algorithm can be easily extended to maintain a window of bitmaps across multiple iterations.

The behavior upon starting migration, also the worst period for downtime, does not match any of these patterns: the number of dirty pages monotonically decreases with iterations and the number of page retransmissions is quite small. Thus, in our implementation we transfer once the whole memory space before enabling the convergence mechanisms. This behavior is similar to the default KVM implementation. Given that this stage can take a long time, a large portion of the migrated pages will be modified by the running applications.

Table I presents the results obtained with our algorithm. The applications are using the Ethernet network while migration proceeds over the InfiniBand network. *Slowdown* presents the impact on end-to-end application performance when running with our implementation. *No convergence* presents the impact on application performance when migration does not converge, i.e. we iterate until application termination. This situation occurs in practice for most of the benchmarks when using the KVM default implementation with the preset downtime of

Algorithm 2 Pseudo code for the proposed mechanism for migration control.

```
1: procedure RDMA_MIGRATE_DIRTY_RAM(migrated : Out, mig_threshold : In, retransmit : In)
2:     for all block_i in ram_list do
3:         for all page_j in block_i do
4:             if block_i is dirty then
5:
6:                 if migrated < mig_threshold then ▷ mig_threshold is initially arbitrarily large
7:                     reset_dirty_bit()
8:                     coalesce(block_i)
9:                     update_retransmit_count()
10:                    update_migrated_count()
11:                    if coalesced_segments > 1 then
12:                        post_rdma_send()
13:                    end if
14:                end if
15:                update_modified_count()
16:            end if
17:        end for
18:    end for
19:    if coalesced_segments ≠ 0 then
20:        post_rdma_send()
21:    end if
22: end procedure
23:
24: procedure RDMA_CONTROL_MIGRATION(retransmitted : In, migrated : In, modified : In, mig_threshold : InOut, timing_info : In)
25:    (a_p, b_p) ← linear_regression(migrated, post_time)
26:    (a_w, b_w) ← linear_regression(migrated, wait_time)
27:    a ← a_p + a_w, b ← b_p + b_w
28:    period ← post_time + wait_time
29:    if period < sampl_period then
30:        new_mig_threshold ← (sampl_period − a)/b
31:    else if migrated < modified then
32:        additional_pages ← (modified − migrated)
33:        max_pages ← (sampl_period − period − a)/b
34:        extra ← min(max_slack_pages, additional_pages)
35:        new_mig_threshold ← mig_threshold + extra
36:    end if
37:    mig_threshold ← (mig_threshold + new_mig_threshold × 3)/4 ▷ Low pass filtering
38:    if migrated history in steady state then
39:        Switch from live migration to downstate
40:    end if
41:    if retransmitted/migrated ≥ 90% then
42:        Switch from live migration to downstate
43:    end if
44: end procedure
```

30ms, which is unattainable in practice.[2] *Downtime* presents the downtime attained by our implementation. By contrast, *downtime_max* is the downtime that results from pure stop-and-copy migration at the start of the application execution, and *downtime_min* is the downtime from stop-and-copy after the application terminates. *Pattern* presents the pattern detected for termination during the application run. All the results were obtained for the scenario where migration is started after the application initialization phase, i.e. in the NAS regions of code used for performance assessments.

As illustrated, our algorithm is able to attain downtimes between 8ms and 520ms for all applications. For all experiments, *downtime min* is below 15ms, but this value occurs after application termination. Unlike our benchmarks, actual scientific applications are long-running and either pure stop-and-copy or live migration is required. The benefits of live migration are clearly demonstrated by our algorithm, which provides downtimes that are orders of magnitude lower than the pure stop-and-copy approach (*downtime max*). Furthermore, for all the applications we evaluated, live migration using our algorithm completely finishes during the first third of application execution.

During any application run, multiple convergence patterns can be matched, depending on the time at which migration has been started. All previous techniques using target downtime as a termination condition are able to match only the equivalent of pattern *b*. The experimental setting, i.e. dedicated high speed network with no rate limit for migration, is the most favorable for achieving a predetermined minimum downtime and implicitly matching pattern *b*. As the results indicate, for a significant number of experiments our algorithm detects termination based on patterns *a* and *c*: **a stronger convergence criteria is clearly needed when increasing VM concurrency or the active dataset**. Furthermore, while all applications have regions where the number of modified pages becomes small (e.g. barriers), these are infrequent and short and steady state is characterized by frequent retransmissions.

VIII. Related work

Virtual machine migration for workstations is thoroughly discussed by Sapuntzakis et al [16]. They present mechanisms for offline migration of multiprogrammed workloads running on single core x86 workstations. In offline migration, VMs are suspended, state is stored in temporary storage (disk or memory) and then transferred to the destination system, where execution is resumed. This technique involves a large downtime and server based applications avoid it to maintain customer satisfaction.

Clark et al [4] discuss live migration of virtual machines using pre-copy techniques for commercial workloads on dual core systems. They introduce the concept of a *writable working set* (WWS) and report low downtimes for their workloads. While the WWS is relatively small (mostly stack pages) for commercial workloads, for scientific workloads it comprises most of the system memory. Voorsluys et al [19] also show an acceptable live migration overhead for data centers serving enterprise-class Internet applications in clouds.

Huang et al [7] present an implementation of pre-copy live migration on Xen using InfiniBand and evaluate it for the NAS parallel benchmarks (MPI) running on clusters using one core per node. We provide the first implementation of live migration in KVM over InfiniBand, while providing a similar level of optimizations, e.g page clustering. They do not report downtimes and indicate that migration impacts the application performance by at most 15%. Our results indicate much larger impacts when increasing the number of cores per VM or the application memory footprint.

Nagarajan et al [15] discuss proactive fault tolerance mechanisms for scientific applications using live migration. They present results for MPI implementations of the NAS benchmarks, on a 16 node cluster with dual core AMD processors and virtual machines constrained to 1GB memory. They also discuss the interaction between health monitoring and migration. Vallee et al [17] also discuss fault tolerance using live migration. While all these studies showcase the promise of pre-copy live migration, as our results indicate, increasing the number of cores and the amount of memory per VM greatly affects performance and more sophisticated techniques for switching from live to stop-and-copy migration are required.

Fault tolerance using migration has also been studied in user level environments such as Adaptive MPI and Charm++ by Chakravorty et al [3] in dual core clusters. Given the increase in core concurrency per node we believe that our results are directly applicable to their work.

Post-copying techniques for live migration are discussed by Hines et al [6] and Moghaddam et al [14]. Their objective is to reduce migration time and they show promising results for commercial workloads. In post-migration or *post-copy*, the remote machine is started and processes are migrated without copying the memory pages, which are copied on demand. The downside of this approach is the significant slowdown that the migrated machine/application might suffer. First touches of a page on the destination system usually result in a network transfer, which has high startup costs. As opposed to *pre-copy* which is bandwidth bound, *post-copy* is a latency bound technique. Post-copy techniques have not been thoroughly evaluated for scientific workloads.

A seemingly unrelated but certainly pertinent area of research is configuring virtualized environments for optimal performance when running scientific applications. In [8] we present an overview and techniques for configuring manycore NUMA nodes. We show that optimal performance is achieved by configurations where cores are partitioned across multiple virtual machines but each VM spans at least a NUMA domain. The core concurrency within a modern NUMA domain is certainly higher than the limits that previous pre-copy techniques can handle.

[2] If we set the downtime threshold for the KVM implementation to be high enough for migration to converge, in most cases the MPI benchmarks abort because of network timeouts.

TABLE I

A SUMMARY OF THE DOWNTIME ACHIEVED BASED ON THE PROPOSED TECHNIQUE AND THE ASSOCIATED SLOWDOWN OF THE APPLICATION. CONVERGENCE TO THE LAST PHASE OF MIGRATION IS ACHIEVED BY EITHER *a*: STABLE MODIFIED LEVEL; *b*: SMALL MIGRATION ACTIVITY; *c*: (90%) RETRANSMISSION OF MODIFIED PAGES.

	Class B, 4 proc VM, OpenMP								**Class B, 4 proc × 2 VM, MPI**							
	bt	**cg**	**ep**	**ft**	**is**	**lu**	**mg**	**sp**	**bt**	**cg**	**ep**	**ft**	**is**	**lu**	**mg**	**sp**
slowdown	17%	40%	22%	16%	67%	12%	19%	20%	46%	52%	66%	56%	31%	31%	53%	57%
no convergence	85%	61%	64%	154%	123%	152%	73%	49%	602%	245%	115%	231%	196%	455%	608%	755%
downtime (ms)	54	10	12	86	8	35	15	75	97	10	11	12	12	85	40	10
downtime max (ms)	8321	7940	7842	8197	7826	7780	7909	7901	7800	8741	7887	10560	8175	7850	11293	11721
downtime min (ms)	10	12	10	5	5	13	5	6	5	6	5	5	7	5	5	5
pattern	*c*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*c*	*b*	*b*	*b*	*b*	*c*	*a*	*b*
	Class C, 8 proc VM, OpenMP								**Class C, 8 proc ×2 VM, MPI**							
slowdown	9%	15%	20%	12%	18%	8%	8%	24%	59%	70%	2%	43%	102%	31%	49%	54%
no convergence	64%	56%	37%	87%	63%	51%	146%	74%	415%	201%	38%	160%	186%	218%	487%	416%
downtime (ms)	90	25	32	465	8	35	320	520	80	25	14	15	29	200	120	93
downtime max (ms)	25675	23205	22910	29856	25100	21910	22778	28407	26305	26743	27155	29243	25743	29343	28743	28743
downtime min (ms)	11	11	14	11	12	11	11	11	11	11	14	11	10	11	11	11
pattern	*a*	*b*	*a*	*a*	*b*	*a*	*c*	*a*	*c*	*a*	*b*	*b*	*b*	*c*	*a*	c

IX. DISCUSSION

Our technique is orthogonal and complementary to existing rate limiting techniques as it only hastens detecting migration termination. Our evaluation is for KVM, which uses a particular open loop control scheme for migration where static rate limits and target downtimes are provided by system administrators. Although Xen [4], [7] and reportedly VMWare, use closed loop techniques with rate limits, these are equivalent to the KVM approach. Xen uses a pre-compiled low and high bandwidth threshold. Migration starts at the low threshold and it is increased in subsequent iterations with a constant additive factor. Live migration is terminated when the bandwidth allocated to migration reaches the high threshold or, only a small pre-compiled number of pages remains to be transferred, which in practice amounts to the KVM target downtime. The net effect of this approach is that Xen will terminate migration when a target downtime is attained or the equivalent of pattern *a* in Figure 10 occurs, in which case it will perform a fixed number of iterations. We believe that our algorithm fits naturally with these closed loop approaches where it will be able to provide the same benefits: lower downtime and lower application performance impact.

Matching any of the patterns discussed earlier does not guarantee attaining the global minimum downtime; our algorithm waits only until downtime becomes relatively small and there are no further improvement expectations. The biggest advantage of this technique is preventing the migration from continuing while degrading the application performance with no clear benefit.

The astute reader has noticed that our evaluation platform, a quad-socket quad-core Intel Tigerton system, has a low memory bandwidth when compared to contemporary systems. This is due to its particular Front Side Bus memory architecture. As memory bandwidth also influences RDMA bandwidth, newer systems might provide a faster draining rate for migration. On the other hand, lower memory bandwidth throttles the memory rate of change in applications. Thus, we believe that our results are valid across any multicore platform.

Even with our improved algorithm we observe a high impact on end-to-end application performance, e.g. 51% average slowdown for NAS class C. This raises the question of whether pure stop-and-copy migration is more appropriate than live migration for scientific applications. For the final version of this paper we plan to provide a comparison of these two approaches. A sound answer to the question requires a very large number of experiments and at the time of this writing we have available only partial data.

X. CONCLUSIONS

In this paper we study the behavior of iterative pre-copy live migration for memory intensive applications and present a detailed performance analysis of the current KVM implementation. Our analysis indicates that for scientific applications, where VMs contain multiple cores, and the application memory rate of change is likely to be higher than the migration draining rate, the existing pre-copy live migration techniques become sub-optimal. We present a novel algorithm that achieves both low downtime and low application performance impact. At the core of the algorithm is detecting memory update patterns and terminating migration when improvements in downtime are unlikely to occur. We implemented this approach in KVM and demonstrated its benefits for both Ethernet and RDMA (InfiniBand) migration. Our technique is complementary to rate limitation techniques and can be easily adopted to other environments, such as Xen. The KVM implementation provides mechanisms very similar to Xen and our performance evaluation provides insights applicable to both environments. Furthermore, given the increase in core concurrency and memory per computation node, we believe that our approach, or similar approaches able to impose fast termination, are mandatory when considering live migration of parallel scientific applications.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. *Technical Report NAS-95-010, NASA Ames Research Center*, 1995.

[3] S. Chakravorty, C. L. Mendes, and L. V. Kal. Proactive fault tolerance in mpi applications via task migration. In *In International Conference on High Performance Computing*, 2006.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. *The 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, pages 273–286, 2005.

[5] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for "autonomic" orchestration. *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 7, 2006.

[6] M. R. Hines and K. Gopalan. Post-copy based Live Virtual Machine Migration using Adaptive Pre-paging and Dynamic Self-ballooning. *The 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, 2009.

[7] W. Huang, Q. Gao, J. Liu, and D. Panda. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. *The 2007 IEEE International Conference on Cluster Computing*, pages 11–20, Sept. 2007.

[8] K. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the Performance of Parallel Applications on Multi-Socket Virtual Machines. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID11)*, 2011.

[9] Kernel Based Virtual Machine. http://www.linux-kvm.org/, 2008.

[10] J. Lange, K. Pedretti, P. Dinda, C. Bae, P. Bridges, P. Soltero, and A. Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of he t2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2011.

[11] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, , and R. Brightwell. Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing. In *In IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[12] National Impact Series: Scientists Look To The Clouds To Solve Complex Questions. Available at http://www.er.doe.gov/News_Information/-News_Room/2009/Oct%2014_ComplexQuestions.html, 2009.

[13] J. Matthews, T. Garfinkel, C. Hoff, and J. Wheeler. Virtual machine contracts for datacenter and cloud computing environments. *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 25–30, 2009.

[14] F. Moghaddam and M. Cheriet. Decreasing live virtual machine migration down-time using a memory page selection based on memory change PDF. *Networking, Sensing and Control (ICNSC), 2010 International Conference on*, pages 355 –359, April 2010.

[15] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. *The 21st annual international conference on Supercomputing*, pages 23–32, 2007.

[16] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[17] G. Vallee, C. Engelmann, A. Tikotekar, T. Naughton, K. Charoenpornwattana, C. Leangsuksun, and S. Scott. A Framework for Proactive Fault Tolerance. *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 659 –664, Mar. 2008.

[18] Virtio: An I/O virtualization framework for Linux. http://www.ibm.-com/developerworks/linux/library/l-virtio/index.html.

[19] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. *Proceedings of the 1st International Conference on Cloud Computing*, pages 254–265, 2009.