Proactive Process-Level Live Migration and Back Migration in HPC Environments

Chao Wang[a], Frank Mueller[a], Christian Engelmann[b], Stephen L. Scott[b]

[a]*Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534*
[b]*Oak Ridge National Laboratory, Computational Sciences and Mathematics Division, Oak Ridge, TN 37831*

Abstract

As the number of nodes in high-performance computing environments keeps increasing, faults are becoming common place. Reactive fault tolerance (FT) often does not scale due to massive I/O requirements and relies on manual job resubmission.

This work complements reactive with proactive FT at the process level. Through health monitoring, a subset of node failures can be anticipated when one's health deteriorates. A novel process-level live migration mechanism supports continued execution of applications during much of processes migration. This scheme is integrated into an MPI execution environment to transparently sustain health-inflicted node failures, which eradicates the need to restart and requeue MPI jobs. Experiments indicate that 1-6.5 seconds of prior warning are required to successfully trigger live process migration while similar operating system virtualization mechanisms require 13-24 seconds. This self-healing approach complements reactive FT by nearly cutting the number of checkpoints in half when 70% of the faults are handled proactively. The work also provides a novel back migration approach to eliminate load imbalance or bottlenecks caused by migrated tasks. Experiments indicate the larger the amount of outstanding execution, the higher the benefit due to back migration will be.

Keywords: Live Migration, Back Migration, Fault Tolerance, High-Performance Computing, Health Monitoring

---

1. Introduction

The current trend in high-performance computing (HPC) is to continually increase the system scale through ever larger number of nodes, each of them consisting of multi-core processors, exceeding the 100,000 processor mark. This substantial system growth poses a challenge in terms of the failure probability for large-scale jobs.

Reliability data of contemporary systems, depicted in Table 1, illustrates that the mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-40 hours,

[*]Principle corresponding author
Email address: `mueller@cs.ncsu.edu` (Frank Mueller)

depending on the maturity / age of the installation [1]. The most common causes of failure are processor, memory and storage errors / failures. When extrapolating for current systems, the mean-time between failures (MTBF) for peta-scale systems is predicted to be as short as 1.25 hours [2] and could be accompanied by one radiation-induced soft error once every 1.5 hours [3]. With current checkpoint/restart (C/R) technology, the wall-clock time of a 100-hour job could well extend to 251 hours due to C/R overhead implying that 60% of cycles are spent on C/R alone [2]. At next-generation extreme scales, future chips are expected to keep failure rates per bit similar to that of current systems [4]. However, the component count will dramatically increase, which leads to a much lower system MTBF.

System	# CPUs	MTBF/I
ASCI Q	8,192	6.5 hrs
ASCI White	8,192	5/40 hrs ('01/'03)
PSC Lemieux	3,016	9.7 hrs
Google	15,000	20 reboots/day

Table 1: Reliability of HPC Clusters

Yet, when fault-tolerant middleware hides such failures and user services remain unaffected, the interpolated fault rate is just over one hour for equivalent number of nodes (c.f. Google in Table 1) [5]. In this spirit, our work focuses on fault-tolerant middleware for HPC systems. More specifically, this paper promotes process-level live migration combined with health monitoring for a proactive fault tolerance (FT) approach that complements existing C/R schemes with self healing whose fault model is based on the work by Tikotekar *et al.* [6].

Health monitoring has recently become a wide-spread feature in commodity and, even more so, in server and HPC components. Such monitors range from simple processor temperature sensors to baseboard management cards (BMCs) with a variety of sensing capabilities, including fan speeds, voltage levels and chassis temperatures. Similarly, the SMART disk standard provides the means to trigger events upon disk errors indicating disk problems, which can be saved in log files or which can trigger exception handling mechanisms. Aided by such capabilities, node failures may be anticipated when the health status of a node deteriorates, *i.e.*, when abnormal sensor readings or log entries are observed.

Health monitoring has been used to model failure rates and, in a reactive manner, to determine checkpoint intervals [7, 8]. In this work, we venture beyond reactive schemes by promoting a *proactive* approach that migrates processes away from "unhealthy" nodes to healthy ones. Such a self-healing approach has the advantage that checkpoint frequencies can be reduced as sudden, unexpected faults should become the exception. This requires the availability of spare nodes, which is becoming common place in recent cluster acquisitions. Current extensions of resource managers are considering (a) a pool of spare nodes as part of a job allocation request and (b) a shared pool of spare nodes between jobs [9]. Nodes from any of these pools can be dynamically requested by a node's MPI daemon, which can then readily utilize our task interpositioning layer to replace a failed node with a spare in a transparent manner to the application. Contention for spare nodes is handled by MPI daemons on the spares that forward requests among each other using a dynamically reshaped radix tree for scalability [10]. This approach is much easier to implement that any active or passive replication scheme as it, in contrast to replication, does not require additional messages prior to fault recovery. Furthermore, the number of spare nodes is typically a small constant while replication strategies may require two times the number of nodes for a given job [11]. Our techniques also generalize to task sharing on a node should not enough spare nodes be available, yet the cost is reduced performance for tasks on such shared nodes. This may result in load imbalance between tasks system-wide resulting in decreased overall performance. Such imbalance might be tolerable when faulty nodes can be brought back online quickly so that processes can migrate back to their original nodes. Other problem sources besides fail-stop node or link failures, such as silent data corruption, e.g., due to radiation-inflicted bit flips, are not considered by this work.

The feasibility of health monitoring at various levels has recently been demonstrated for temperature-aware monitoring, *e.g.*, by using ACPI [12], and, more generically, by critical-event prediction [13]. Particularly in systems with thousands of processors, fault handling becomes imperative, yet approaches range from application-level and runtime-level to the level of OS schedulers [14, 15, 16, 17]. These and other approaches differ from our work in that we promote *live migration combined with health monitoring*.

Contributions: We have designed an automatic and transparent mechanism for *proactive* FT of arbitrary MPI applications. The implementation, while realized over LAM/MPI's C/R support [18] through Berkeley Labs C/R (BLCR) [19], is in its mechanisms applicable to any process-migration solution, *e.g.*, the Open MPI FT mechanisms [20, 21]. BLCR is an open source, system-level C/R implementation integrated with LAM/MPI via a callback function. The original LAM/MPI+BLCR combination [22] only provides reactive FT and requires a complete job restart from the last checkpoint including job resubmission in case of a node failure. Recent work enhances this capability with a job pause/continue mechanism that keeps an MPI job alive while a failed node is replaced by a spare node [23]. Paused, healthy tasks are rolled back to and spare nodes proceed from the last checkpoint in a coordinated manner transparent to the application.

Another contribution of this paper is to avoid roll-backs to prior checkpoints whenever possible. By monitoring the health of each node, a process is migrated as a precaution to potentially imminent failure. To reduce the impact of migration on application performance, we contribute a novel *process-level live migration* mechanism as an enhancement to the Linux BLCR module. Thus, execution proceeds while a process image is incrementally and asynchronously transferred to a spare node. This reduces the time during which the process is unresponsive to only a short freeze phase when final changes are transferred to the spare node before re-activating execution on the target node. Hence, MPI applications execute during much of process migration. In experiments, we assessed the trade-off between lower end-to-end wall-clock times of jobs subject to live migration *vs.* the slightly prolonged duration for migration as opposed to a traditional process-freeze approach. Depending on the estimated remaining up-time of a node with deteriorating health, one can choose between live and frozen migration schemes.

Our results further demonstrate that proactive FT complements reactive schemes for long-running MPI jobs. Specifically, should a node fail without prior health indication or while proactive migration is in progress, our scheme reverts to reactive FT by restarting from the last checkpoint. Yet, as proactive FT has the potential to prolong the mean-time-to-failure, reactive schemes can lower their checkpoint frequency in response, which implies that proactive FT can lower the cost of reactive FT. More specifically, experimental results indicate that 1-6.5 seconds of prior warning are required to successfully trigger live process migration while similar operating system (OS) virtualization mechanisms require 13-24 seconds. The approach further complements reactive FT by allocating nearly twice as long a checkpointing interval due to proactive migration when 70% of the failures are predicted only a few seconds prior (derived from [13]).

A migrated task could present a bottleneck due to (1) increased hop counts for communication from/to the spare node, (2) reduced resources in heterogeneous clusters (lower CPU/memory/network speed), or (3) placement of multiple MPI tasks on a node if not enough spare nodes are available. We contribute **back migration** as a novel methodology. We have implemented the back migration mechanism within LAM/MPI and BLCR based on process-level live migration in reverse direction. Our results indicate that we can benefit from back migration when, on average, 10.19% of execution is still outstanding just for our set of benchmarks. For larger applications, benefits are projected to occur for even smaller remaining amounts.

2. Design

The focus of this work are the design, implementation and evaluation of process-level live migration mechanisms, including back migration. To this end, we assume the availability of spare nodes as migration targets. We also note that globally coordinate resilience schemes have proved to be feasible at scale as interconnects tend to provide scalable interconnects, such as Cray's hypercube topologies, or even special collective interconnects, such as BlueGene's trees for collectives and barriers (interrupt tree) [24]. This property will be exploited for draining messages across nodes during late phases of migration. We further assume the availability of health monitoring with process-level notification [25]. The technical details and overheads of health monitoring are beyond of scope of this work, but we note that monitored health data tend to result in predictions in the order of tends of minutes (fan speeds, temperatures) if not hours or days (hard drive logs via SMART) in practice [26, 27]. As will be shown, such predictions are more than sufficient to accommodate proactive actions.

Figure 1 depicts the system components and their interaction, *i.e.*, the chronological steps involved in process migration of each MPI job and job dependencies with data exchanges. In the following, we discuss system support for live migration at two levels: (1) the synchronization and coordination mechanisms within an MPI job and (2) live migration with incremental update support at the process/MPI task level. We further consider the tradeoff between live and frozen migration options and develop approaches to manage "dirty" memory, *i.e.*, memory written since the last incremental update. We then analyze the mechanism for and assess the benefits of back migration.

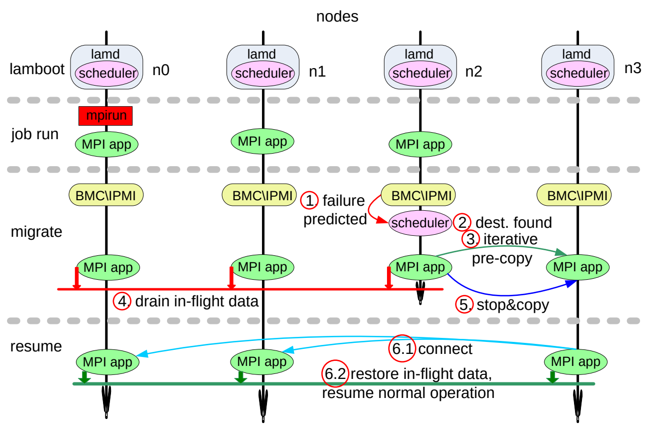


Figure 1: Job Live Migration

2.1. Live Migration at the Job Level

Figure 1 depicts steps 1-6, each of which are described in the following.

Step 1: Migration Trigger: In our system, the per-node health monitoring mechanism is realized on top of a Baseboard Management Controller (BMC). It is equipped with sensors to monitor different properties, *e.g.*, sensors providing data on temperature, fan speed, and voltage. We also employ the Intelligent Platform Management Interface (IPMI), an increasingly common management/monitoring component that provides a standardized message-based interface to obtain sensors readings for health monitoring. We further designed a decentralized scheduler, which can be deployed as a stand-alone component or as an integral process of an MPI daemon, such as the LAM daemon (lamd). The scheduler will be notified upon deteriorating health detected by BMC/IPMI, *e.g.*, due to a sensor reading exceeding a threshold value.

Step 2: Destination Node Determination: When the scheduler component on the health-decaying node receives the early warning issued in step 1, it first chooses a spare node as a replacement for the unhealthy node. For the current petascale and next-generation exascale system with tens of thousands of nodes [28], allocating spare nodes for fault tolerance turns to be inexpensive. We expect that job schedulers may soon simply allocate a limited number of spares as a default or upon request during job submission. As of now, we still have to explicitly over-provision by setting a constant number of nodes aside during job submission. While performance impacts due to different task placement after spare node utilization is beyond the scope of this paper, we do note that current job scheduler do not perform communication-sensitive task placement for specific interconnect topologies in first place. Furthermore, our back migration scheme (see Section 2.4 would restore original placements and thereby address any future communication-sensitive placement.

These spare nodes comprise the destination for process migration. An alternative to spare nodes is provisioning, which tends to result in benefits only when a faulty node can be rebooted/replaced quickly or if an application run is close to completion. In over-provisioning, we choose the most lightly loaded node as a migration destination if its available memory space suffices for two task, in which case the node doubles for two MPI tasks. Sharing nodes may result in imbalance due to bearing twice the load of regular nodes, which generally results in lower overall application performance. Such imbalance might be tolerable when faulty nodes can be brought back online quickly so that processes can migrate back to their original nodes (see back migration in Section 2.4). Nonetheless, higher sustained application performance can be guaranteed when unloaded spares are available as migration targets.

Step 3: Memory Precopy: Once a destination node is determined, the scheduler initiates the migration of a process on both destination and source nodes. The objective of the first stage of migration is to transfer a memory snapshot of the process image to the spare node, yet to allow the application to execute during this stage, hence the name *live* migration. The transfer of the process image occurs at page granularity and is repeated for pages written to by the application between image scans. During the first process image scan (first iteration), all non-zero pages are transferred from the source node to the destination node. On subsequent scans/iterations, only the pages updated since the previous scan are transferred. When the number of such "dirty" pages between scans does not change significantly anymore, the scan loop terminates. System support for tracking dirty pages is discussed later in the context of memory management.

Step 4: In-flight Message Drainage: Before we stop the process and migrate the remaining dirty pages with the corresponding process state to the destination node, all MPI tasks need to coordinate to reach a consistent global state. Based on our LAM/MPI+BLCR design, message passing is dealt with at the MPI level while the process-level BLCR mechanism is not aware of messaging at all. Hence, we employ LAM/MPI's job-centric interaction mechanism for the respective MPI tasks to clear in-flight data in the MPI communication channels.

Step 5: Stop&Copy: Once all the MPI tasks (processes) reach a consistent global state, the process on the source node freezes (suspends) application execution but still copies the remaining dirty pages (written to since the last iteration in step 3) and the final process state (registers, signal information, pid, files etc.) to the destination node. All other MPI tasks are suspended at their point of execution.

Step 6: Connection Recreation, Messages Restoration and Job Continuation: When the process is ready on the destination node, it sets up a communication channel with all other MPI tasks. Subsequently, the drained in-flight messages are restored, and all the processes resume execution from the point of suspension.

Discussion: Our approach is conservative in draining messages of all MPI tasks (step 4). It would be sufficient to only drain messages of the subset of tasks that communicate with the process subject to migration. However, this would require other tasks to receive two notifications, one on the migration intent and another on migration completion. Nonetheless, the migrating node would only proceed after receiving responses from all others that their messages are drained. In our experiments, the migrating node was always the bottleneck. Hence, any attempts to allow other nodes to proceed without global coordination are likely to not result in any benefit since they are not on the critical path.

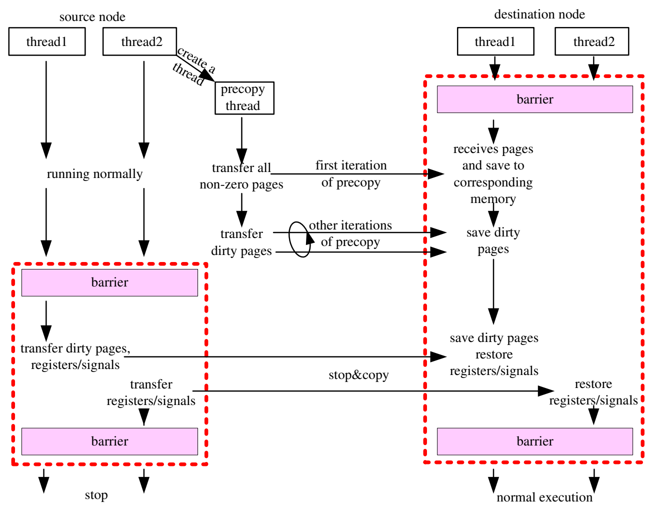

source node
thread1
thread2
create a thread
precopy thread
running normally
transfer all non-zero pages
first iteration of precopy
transfer dirty pages
other iterations of precopy
barrier
transfer dirty pages, registers/signals
transfer registers/signals
stop&copy
barrier
stop
destination node
thread1
thread2
barrier
receives pages and save to corresponding memory
save dirty pages
save dirty pages restore registers/signals
restore registers/signals
barrier
normal execution

Figure 2: Process Migration with Precopy (Kernel Mode in Dotted Frame)

2.2. Live Migration at the Process Level

The incremental precopy and stop&copy (steps 3 and 5 in Figure 1) are performed at the process level involving only the destination and source nodes. Yet, there are trade-offs between simply stopping an application to engage in a frozen copy and the alternative of a live precopy with continued application progress. The latter, while generally resulting in shorter overall application wall-clock time, comes at the expense of background network activity, possibly even repeatedly transmitting dirtied pages. This also raises the question when the iterative precopy loop should terminate.

Figures 2 and 3 show migration with precopy (live) and without (stop&copy-only). Compared to a traditional job resubmission resuming execution from the last checkpoint in a reactive FT scheme, both of these proactive migration schemes lower the expected execution time of the MPI application in the presence of failures. This is especially the case for HPC environments where the MTBF is low, which typically implies that the number of compute nodes is high, the run-time of an application is long, and the memory footprint is large.

One of our objectives is to reduce the aggregate downtimes over all nodes, *i.e.*, the duration of the stop&copy step should be small. Live migration with incremental precopy results not only in shorter downtime on the local node (to transfer dirty pages plus other process state) but also in reduced suspension of all other nodes (once MPI message queues are drained) since fewer pages remain dirty after precopy. Another objective is to tolerate the imminent fault. The shorter the migration duration, the higher the probability that our proactive scheme makes a reactive restart from a prior checkpoint unnecessary. Frozen migration consisting only of the stop&copy step takes less overall time during migration, thereby increasing chances for successful migration. A compromise might even be to stop the precopy step prematurely upon receipt of another fault event indicating higher urgency of the health problem.

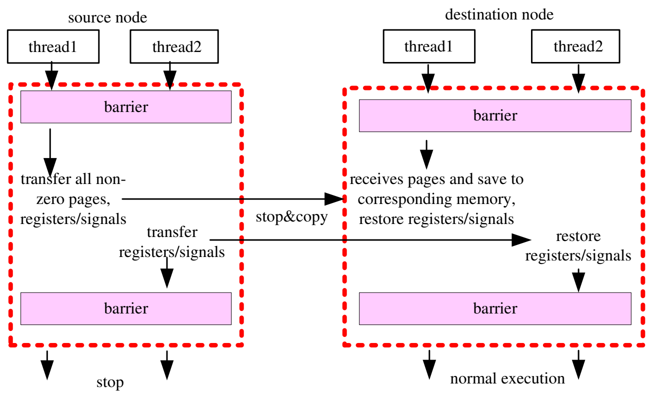

source node
thread1
thread2
barrier
transfer all non-zero pages, registers/signals
transfer registers/signals
barrier
stop
stop&copy
destination node
thread1
thread2
barrier
receives pages and save to corresponding memory, restore registers/signals
restore registers/signals
barrier
normal execution

Figure 3: Process Migration without Precopy (Kernel Mode in Dotted Frame)

Two major factors affect the tradeoff between the downtime and the migration duration. First, the network bandwidth shared between the MPI job and the migration activity is limited. If an application utilizes less network bandwidth, more bandwidth can be consumed by the migration operation. Thus, the precopy step may only have minimum impact on application progress. More communication-intensive applications, however, may leave less unused bandwidth for live migration (during precopy). This could both slow down a local MPI task (with potentially globally unbalanced progress due to communication) and prolong precopy duration, ultimately requiring a premature termination of this step. High-end HPC installations may provide multiple network backbones to isolate MPI communication from I/O traffic, the latter of which covering checkpointing and migration traffic as well. In such systems, bandwidth issues may not be as critical (but cannot be fully discounted either).

Second, the page write rate (dirtying rate) affects the trade-off between downtime and migration duration. The dirtying rate is affected by the memory footprint (more specifically, the *rewrite* working set) of the application. A larger number of pages repeatedly written within a tight loop will prolong the precopy duration, which might lower the probability for successful migration (compared to non-live migration). In fact, the page access patterns of the studied benchmarks will be shown to differ significantly. We further discuss this issue in the experimental section.

2.3. Memory Management

A prerequisite of live migration is the availability of a mechanism to track modified pages during each iteration of the precopy loop. Two fundamentally different approaches may be employed, namely page protection mechanisms or page-table dirty bits. Different implementation variants build on these schemes, such as the bookkeeping and saving scheme that, based on the dirty bit scheme, copies pages into a buffer [29].

Under the page protection scheme, all writable pages of a process are write-protected before each iteration occurs. When a page is modified (written to for the first time), a page fault exception is raised. The triggered exception handler enables write access to the page and records the page as dirty. Upon return from the handler, the write access is repeated successfully, and subsequent writes to the same page execute at full speed (without raising an exception). At the next iteration, pages marked as dirty are (1) write protected again before being (2) copied to the destination node. This ordering of steps is essential to avoid races between writes and page transfers.

The dirty bit scheme is implemented by duplicating the dirty bit of the page-table entry (PTE). The dirty bit is set in response to handling by the memory management unit (MMU) whenever the first write to the page is encountered. At each iteration, the duplicate bit is checked and, if set, is first cleared before the page is transferred to the destination node. To provide this shadow functionality, kernel-level functions accessing the PTE dirty bit are extended to also set the duplicate bit upon the first write access.

The page protection scheme has certain draw-backs. Some address ranges, such as the stack, can only be write protected if an alternate signal stack is employed, which adds calling overhead and increases cache pressure. Furthermore, the overhead of user-level exception handlers is much higher than kernel-level dirty-bit shadowing. Thus, we selected the dirty bit scheme in our design. Implementation-specific issues of this scheme are discussed in detail in the next section.

2.4. Back Migration

To determine if we can benefit from back migration before actually migrating back an MPI task when the original node is recovered, the performance of MPI tasks across all nodes must be monitored. Systems, such as Ganglia [30], htop [31] and PAPI [32], monitor the performance of entire nodes rather than MPI task-specific performance or even timestep-specific metrics. Per-node measurements tend to be inaccurate as they fail to capture the “velocity” of an MPI job *before and after* task migration. In our design, we eliminate node-centric draw-backs through self-monitoring within each MPI task.

HPC applications spend 90 percent of the execution time of programs in loops, and “loop profiling tools” can be used to gather timestep information [33]. During a job’s execution, MPI tasks record the duration of a timestep and relay this information to the same decentralized scheduler that we designed for migration trigger as shown in Figure 1. The scheduler compares the “velocity” of the MPI job *before and after* task migration to decide whether or not to migrate an MPI task back to the original node once this node is brought back online in a healthy state. The decision considers (a) the overhead of back migration and (b) the estimated time for the remaining portion of the job, which is also recorded for the MPI job and communicated between the job and the scheduler. We assume back migration overhead to be symmetric to the initial migration overhead.

3. Implementation

We next provide implementation details of our process-level live migration under LAM/MPI+BLCR. The overall design and the principle implementation methodology are applicable to arbitrary MPI implementations, *e.g.*, Open MPI and MPICH.

3.1. Failure Prediction and Decentralized Scheduler

As outlined in Section 2, the capabilities of the BMC hardware and IPMI software abstraction were utilized by our framework to predict failures. In the following, we focus on the system aspects of the live migration mechanism to tolerate imminent faults and only to a lesser extent on fault prediction, a topic mostly beyond the scope of this paper.

The scheduler is implemented as a process of lamd communicating through the out-of-band channel provided by LAM/MPI. When the scheduler daemon is notified by the BMC/IPMI health monitoring component, it consults the database to retrieve information about the MPI jobs and the nodes in the LAM universe. A spare node or, alternatively, the most lightly loaded node is chosen as the migration target. In our implementation, we also check the BMC/IPMI sensor readings with preset thresholds to determine if the degree of urgency allows accommodation of a precopy step. For example, if the temperature is higher than a high watermark (indicating that failure is almost instantly imminent), the frozen migration (stop&copy-only) is launched, which guarantees a shorter migration duration. However, if the value is closer to the low watermark (indicating that failure is likely approaching but some time remains), the live migration (with its precopy step) is launched, which reduces overall application downtime. The design allows additional schemes, selected through a combination of sensor and history information, to be integrated as plug-ins in the future.

Next, we present implementation details of the process-level mechanisms, including dirty page tracking and process image restoration, as they were realized within BLCR. This is followed by the MPI-level implementation details based on the fundamental capabilities of BLCR, including the maintenance of consistency of a global state for the entire MPI job, as realized within LAM/MPI.

3.2. Process-Level Support for Live Migration

As part of our implementation, we integrated several new BLCR features to extend its process-level task migration facilities and to coordinate LAM/MPI's callback mechanism amongst all MPI processes. The scheduler discussed previously issues the *cr_save* and *cr_restore* commands on the source and destination nodes, respectively, when it deems live migration with precopy to be beneficial. Subsequently, once the precopy step has completed, LAM/MPI issues the *cr_stop* and *cr_quiesce* commands on the source node and all other operational nodes, respectively.

3.2.1. Precopy at the Source Node: cr_save

The *cr_save* command implements a sequence of steps specific to our live migration. It first sends a signal to the process on the source node where it triggers the creation of a new thread. This thread performs the precopy work at the user level. We chose a solution at the user level since a kernel-level precopy can block application threads, thereby hampering overall computational progress. During the first iteration of the precopy loop, all non-empty pages are transferred to the destination node. Subsequent iterations, in contrast, result in transfers of only those pages modified since the last iteration. The top half of Figure 2 at the source node depicts this procedure. Recall that the dirty bit approach implemented in our system tracks if a page has been written to since the last transfer (iteration).

The Linux kernel maintains a dirty bit in the PTE. It is automatically set by the hardware (MMU) when the page is modified. Yet, we cannot utilize this dirty bit for our purposes since its state is tightly coupled with the Linux memory management, specifically the swap subsystem. We have to clear such a flag for each iteration of our page sweep, yet clearing the PTE dirty bit would indicate that a page need not be written on a subsequent swap-out, which is violating consistency. Instead of modifying the dirty bit semantics, we decided to shadow the dirty bit within the reserved bits of PTE. Updates to this shadow dirty bit may occur through system functions without much overhead (compared to user-level page protection).

This shadow dirty bit still needs to be preserved across swapping, which is a challenge since swapped PTEs are partially invalidated. One solution would be to simply disable swapping, which might be an option for HPC, but a more general solution is preferable. Linux actually preserves selected fields of a swapped out PTE, among other things to associate with it the swap disk device and the offset into the device where the page resides. This swap information includes a set of bits that are preserved, one of which we utilized as the shadow dirty bit. We implemented this approach in x86_64 and i386 Linux kernels similar to an orthogonal IA64 design by HP [34]. An alternative to the dirty bit approach is to use page protection, which is already supported in POSIX-compliant kernels. This alternative has been investigated [35] but recent comparisons indicate considerably higher costs than the dirty bit approach [36]. Our on-going work includes integration of our techniques in Open MPI and BLCR for various Linux distributions with options for kernel patching or, alternatively, kernel module extensions for dirty bit support.

During live migration, a precopy thread iteratively transfers dirty pages to the destination node until one of the following conditions is satisfied:

- The aggregate size of dirty memory during the last iteration is less than a lower *memory threshold* (default: 1MB);
- the aggregate difference of transferred memory in consecutive iterations is less than a small *difference threshold* (indicating that a steady state of the rate in which pages are written to is reached);
- the remaining time left before the expected failure is below a lower *overhead threshold*.

As shown in Figures 5a and 5b, the page dirty modification eventually stabilizes or its fluctuation is regular (repeats periodically). We could support an empirical factor in the algorithm to keep a profile history of the page modification rate and its regularity. Such model parameters could steer future application runs during live migration / precopy in choosing a more sensitive termination condition. This could be especially beneficial for jobs with long runtime (a large number of timesteps) or repeated job invocations.

Once the precopy step terminates, the thread informs the scheduler, who then enters the stop&copy step in a coordinated manner across all processes of the MPI job. This includes issuing the *cr_stop* command on the source node.

3.2.2. Freeze at the Source Node: cr_stop

The redesigned *cr_stop* command signals the process on the source node to freeze execution, *i.e.*, to stop and copy the pages dirtied in the last iteration of the precopy step to the destination node. Threads of the process subsequently take turns copying their own state information (registers, signal information, etc.) to the destination node. This functionality is performed inside the Linux kernel (as an extension to BLCR) since process-internal information needs to be accessed directly and in a coordinated fashion between threads. The lower half of Figure 2 depicts these actions at the source node.

Between the precopy and freeze steps, the processes of an MPI job also need to be globally coordinated. The objective is to reach a consistent global state by draining all in-flight messages and delaying any new messages. This is accomplished in two parts at the source node, implemented by the *cr_save* and *cr_stop* commands. At the destination node, a single command implements the equivalent restore functionality.

3.2.3. Precopy and Freeze at the Destination Node: cr_restore

At the destination node, the *cr_restore* command with our extensions is issued by the scheduler. This results in the immediate creation of an equal number of threads as were existent on the source node, which then wait inside the Linux kernel for messages from the source node. A parameter to the command, issued by the scheduler, indicates whether or not a live/precopy step was selected on the source node. In either case, one thread receives pages from the source node and places them at the corresponding location in local memory. All threads subsequently restore their own state information received from the corresponding source node threads, as depicted in Figures 2 and 3 at the destination node. After the process image is fully transferred and restored, the user mode is entered. Next, the MPI-level callback function is triggered, which creates the connections with the other MPI processes of the compute job and restores the drained in-flight messages discussed next.

3.2.4. Process Quiesce at Operational Nodes: cr_quiesce

In steps 4 and 6 of our design (cf. Section 2), processes on all other operational nodes drain the in-flight messages before the stop&copy step. They then remain suspended in this step, creating a connection with the new process on the destination (spare) node and ultimately restoring in-flight messages after. This sequence of actions is triggered through the newly developed *cr_quiesce* command. In our implementation, issuing this command signals the process (MPI task), which subsequently enters the MPI-level callback function to drain the messages, waits for the end of the stop&copy step of the faulty/spare nodes, and then restores its communication state before resuming normal execution.

3.3. Job Communication and Coordination Mechanism for Live Migration

In our implementation, we extended LAM/MPI with fundamentally new functionality provided through our BLCR extension to support live migration of a process (MPI task) within a job. Our approach can be divided into four stages introduced in their respective temporal order:

Stage 1: Live/Precopy: The scheduler of the LAM daemon, lamd, determines whether or not live/precopy should be triggered. If sufficient time exists to accommodate this stage, the *cr_save* and *cr_restore* commands are issued on the source and destination nodes, respectively. During the precopy step, the compute job continues to execute while dirty pages are sent iteratively by a precopy thread on the source node. If time does not suffice to engage in this live/precopy stage, the next stage is entered immediately.

Stage 2: Pre-Stop&Copy: In this stage, the scheduler issues a stop&copy command for *mpirun*, the initial LAM/MPI process at job invocation, which subsequently invokes the *cr_stop* and *cr_quiesce* commands on the source node and any other operational nodes, respectively. Once signaled, any of these processes first enters the callback function, which had been registered as part of the LAM/MPI initialization at job start. The callback forces a drain of the in-flight data to eventually meet a consistent global state for all processes (MPI tasks).

Stage 3: Stop&Copy: Upon return from the callback function, the process on the source node stops executing and transfers the remaining dirty pages and its process state. Meanwhile, the processes on other operational nodes suspend in a waiting pattern, as discussed previously.

Stage 4: Post-Stop&Copy: Once all the state is transferred and restored at the destination (spare) node, the process is activated at this node and invokes the LAM/MPI callback function again from the signal handler. Within the callback handler, the migrated process relays its addressing information to *mpirun*, which broadcasts this information to all other processes. These other processes update their entry in the local process list before establishing a connection with the migrated process on the spare. Finally, all processes restore the drained in-flight data and resume execution from the stopped state.

3.4. Back Migration

Our approach requires that MPI tasks perform self-monitoring of execution progress along timesteps. One solution is to provide an interposition scheme that intercepts selected runtime calls of an application at the PMPI layer similar to mpiP [37], a lightweight profiling library for MPI applications to collect statistical information about MPI functions. However, to estimate the benefit of the potential back migration, we must need information about the number of active MPI tasks, which the MPI standard does not natively support. Thus, our implementation provides an API with calls that can be directly added to MPI application code within the timestep loop. These called routines subsequently communicate with the decentralized scheduler to allow timestep-centric bookkeeping of an MPI task's progress. Our back migration approach can be divided into three stages described in the following.

Stage 1: Pre-Live Migration: Instrumentation is added around the time step loop of the MPI application to measure the overhead of each time step. This information is subsequently sent together with the bound on the total number of time steps and the current time integration number to the scheduler of the LAM daemon, which records the last information it received.

Stage 2: Live Migration: When a health problem is predicted to result in a future node failure so that live migration is triggered, the scheduler measures and records the migration overhead.

Stage 3: Post-Live Migration: Just as prior to live migration, the application and scheduler continue to assess the overhead of time steps. The scheduler further compares the difference in "velocity" of the MPI job *before and after* the migration and compares it with the migration overhead to decide whether or not to migrate an MPI task back to the original node once this node is brought back online in a healthy state.

4. Experimental Framework

Experiments were conducted on a dedicated Linux cluster comprised of 17 compute nodes, each equipped with two AMD Opteron-265 processors (each dual core) and 2 GB of memory. The nodes are interconnected by two networks, both 1 Gbps Ethernet. The OS used is Fedora Core 5 Linux x86_64 with our dirty bit patch as described in Sections 2 and 3. One of the two networks was reserved for LAM/MPI application communication while the other supported process migration and other network traffic. In the next section, we present results for experiments that exploited one network interconnect for MPI communication and another for migration. As discussed in Section 2.2, the MPI job and the migration activity may compete for network bandwidth if only a single network is available. It is common for high-end HPC to install two separate networks, one reserved for MPI communication and the other for operations such as I/O, booting, system setup and migration. In our experiments, we also assessed the system performance with a single network responsible for both MPI communication and migration activities. The results are not significantly different from those with two networks, which shows that for the applications evaluated in our system communication and memory intensity do not coincide. The cost of performing migration over the same interconnect as communication does not change for a single interconnect as we drain messages and then freeze communication during migration. Furthermore, for larger systems with higher bandwidth interconnects (1Gbps Ethernet vs. 40Gbps QDR IB [38]), the bandwidth gain is offset by increases in RAM sizes that applications utilize. Though our experiments are performed with Ethernet, our implementation is transparent to the network (including Infiniband), which is managed at the MPI layer. For example, [39] provides a migration framework for MPI over InfiniBand. Our experiments cite a cost of 1-6.5 seconds for one GB of data over Ethernet. An extrapolation to 32GB of RAM utilizing 50% of Infiniband's quad data rate (QDR) bandwidth would imply a cost of 1.6-3.25 seconds, which shows that our results are realistic even for large HPC systems. For the future work, we will create and assess applications with varying communication rates and memory pressure to measure the tradeoff between live and frozen migrations and to provide heuristics accordingly, as discussed in the next section.

We have conducted experiments with several MPI benchmarks. Health deterioration on a node is simulated by notifying the scheduler daemon, which immediately initiates the migration process. To assess the performance of our system, we measure the wall-clock time for a benchmark with live migration, with stop&copy-only migration and without migration. The migration overheads are introduced by transferring the state of the process, including the dirty pages, and the coordination among the MPI tasks. In addition, the actual live migration duration can be attributed to two parts: (1) the overhead incurred by the iterative precopy and (2) the actual downtime for which the process on the source node is stopped. Accordingly, precopy durations and downtimes are measured.

We do not have access to larger clusters that allow us root access to install kernel modules, which is required for the experiments. Experiments on our 17-node cluster indicate that running 2 MPI processes per node (one per socket) to simulate a larger job on a larger cluster skews results in that intra-node MPI communication is performed via shared memory, which results in much lower latencies and even improves our results (for migration within a node), yet may not be reflective of larger clusters that have real spare nodes to migrate to. Using multiple cores per socket skews results even further. Our algorithmic design features scalable data structures, such as a radix tree representation of nodes [25].

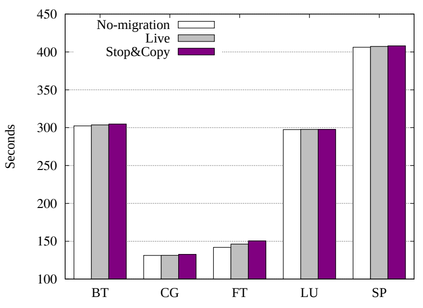
5. Experimental Results

Experiments were conducted to assess (a) overheads associated with the migration, (b) frozen and live migration durations, (c) scalability for task and problem scaling of migration approaches, (d) page access patterns and migration times, and (e) back migration conditions.

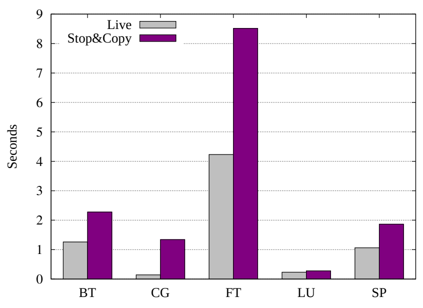
Results were obtained for the NAS parallel benchmarks (NPB) version 3.2.1 [40], a suite of programs widely used to evaluate the performance of parallel systems. Out of the NPB suite, the BT, CG, FT, LU and SP benchmarks were exposed to class B and class C data inputs running on 4, 8 or 9 and 16 nodes. Some NAS benchmarks have 2D, others have 3D layouts for 2^3 or 3^2 nodes, respectively. In addition to the 16 nodes, one spare node is used, which is omitted (implicitly included) in later node counts in this paper. The NAS benchmarks EP, IS and MG were not suitable for our experiments since they execute for too short a period to properly gauge the effect of imminent node failures. (With class B data inputs, completion times for EP, IS and MG are 11.7, 4.8 and 4.1 seconds, respectively, for 16-node configurations.) The characteristics of the NAS PB in terms of computation [40] and communication [41] are discussed in literature. CG and BT are computation bound, FT is communication bound while SP and LU balance communication and computation. CG and LU have low frequencies of writes to disjoint pages (high spatial locality) while SP, BT and FT perform increasing number of disjoint page writes, as reflected by the required memory transfer rates for live migration.

5.1. Migration Overhead

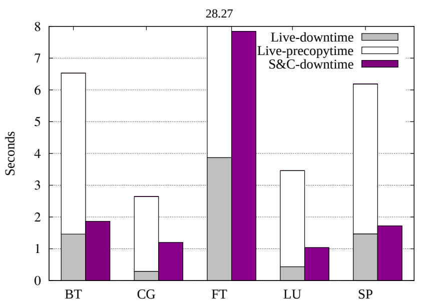
The first set of experiments assesses the overhead incurred due to one migration (equivalent to one imminent node failure). Figure 4a depicts the job execution time without any migration, with live migration and with frozen (stop&copy-only) migration under class C inputs on 16 nodes. The corresponding overheads per scheme are



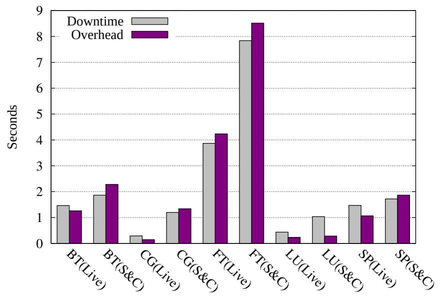
(a) Job Execution Time for NPB C-16



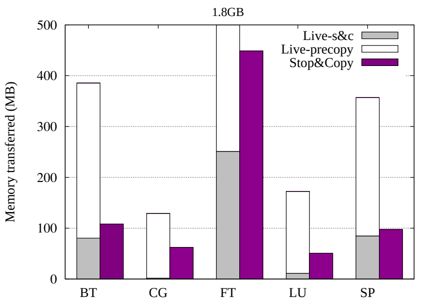
(b) Migration Overhead for NPB C-16



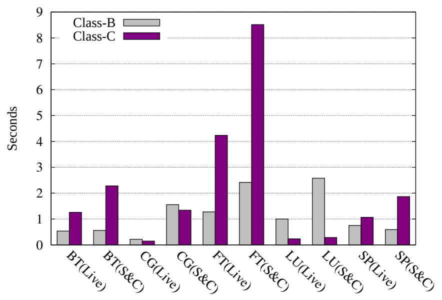
(c) Migration Duration for NPB C-16



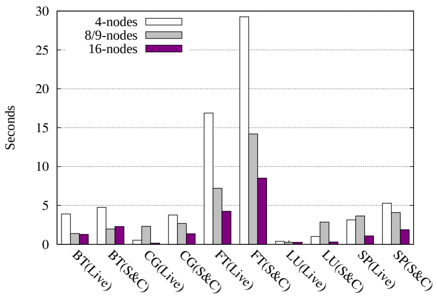
(d) Downtime vs. Overhead for NPB C-16



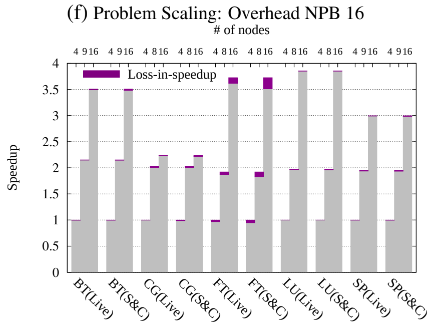
(e) Memory Transferred for NPB C-16



(f) Problem Scaling: Overhead NPB 16



(g) Task Scaling: Overhead of NPB C



(h) Speedup for NPB Class C

Figure 4: Evaluation with NPB for Live Migration (C-16: Class C on 16 Nodes)

depicted in Figure 4b. The results indicate that the wall-clock time for execution with live migration exceeds that of the base run by 0.08-2.98% depending on the application. The overhead of frozen migration is slightly higher at 0.09-6%. In contrast to most benchmarks with small memory footprints, FT illustrates the trend for larger memory utilization: Live migration results in much lower overhead than frozen migration. These savings will only increase for larger memory utilizations. The largest additional cost, 6.7% for frozen migration, was observed for FT under class B inputs for 16 nodes (not depicted here) due to its comparatively large memory footprint (113 MB) and relatively short execution time (37 seconds). Amongst BT, CG, FT, LU and SP under classes B and C running on 4, 8 or 9 and 16 nodes, the longest execution time is 20 minutes (for SP under class C inputs on 4 nodes, also not depicted here). Projecting these results to even longer-running applications, the overhead of migration becomes less significant (if not even insignificant) considering the equivalent checkpointing overhead under current mean-time-to-failure (MTTF) rates.

5.2. Migration Duration

Besides overhead due to migration, we assessed the actual migration duration including live migration (precopy duration) and the downtime of other nodes during migration schemes. Figure 4c depicts (a) live migration with the precopy and downtime fractions and (b) frozen migration with only its downtime during the stop&copy step, both for NPB with class C inputs on 16 nodes. The precopy duration was measured from the issuing of the *cr_save* command to its completion. The stop&copy downtime was measured from issuing *cr_stop* on the source node / *cr_quiesce* on the operational nodes to the resumption of the job execution. Live downtime ranged between 0.29-3.87 seconds (0.14-2.73% of the execution time) while stop&copy downtime was between 1.04-7.84 seconds (0.35-5.49% of the execution time). Live migration pays a penalty for the shorter downtime in that its precopy duration is prolonged. Precopy adds another 2.35-24.4 seconds (1.02-17.18% of the execution time). Nonetheless, the precopy stage does not significantly affect execution progress of the application as it proceeds asynchronously in the background. Figure 4d illustrates this fact for each scheme by comparing the downtime (from Figure 4b) with migration overhead for both frozen (stop&copy) and live migration. Both schemes show a close match between their respective downtime and overhead numbers. (Some benchmarks show shorter overhead than the absolute downtime due to larger variances in job execution times subsuming the shorter overheads, see Section 5.3.)

Figure 4e depicts the amount of memory transferred during migration. With frozen (stop&copy-only) migration, memory pages of a process cannot be modified while the process remains inactive in this stage. In contrast, live migration allows pages to be modified and consequently requires repeated transfers of dirty pages. Hence, both precopy duration and downtime are a function of the write frequency to disjoint memory pages. Frozen (stop&copy-only) migration results in larger memory transfers (50.7-448.6MB) than just the stop&copy step of live migration (1.7-251MB), yet the latter incurs additional transfers (127.4-1565MB) during the precopy step. This result is consistent with the downtime observations of the two schemes discussed above. Our experiments also indicate an approximate cost of 0.3 seconds for MPI-level coordination and communication during live migration plus a cost of less than 0.1 seconds transferring process state information, *e.g.*, registers and signal information. Hence, both the precopy duration and the downtime are almost entirely due to the overhead of transferring memory pages. Hence, the overall trends and patterns of Figures 4c and 4e tend to be similar.

Experiments with two simultaneous migrations indicated that the overall cost is slightly lower than the cumulative costs of two migrations in sequence. This is due to the availability of full bisectional bandwidth of our interconnect for the same data transfer overhead but slightly lower synchronization overhead since simultaneous migrations share a message drainage barrier.

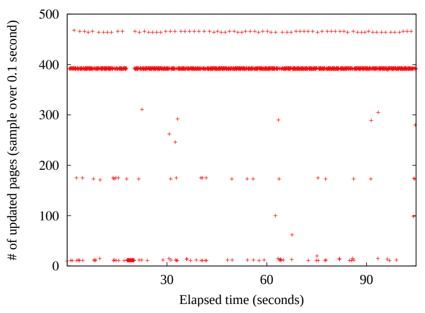
5.3. Effect of Problem Scaling

Figure 4f depicts the effect of migration on scaling the problem size with class B and C inputs on 16 nodes. For BT, FT and SP, we observe an increase in overhead as the task size increases from class B to class C. This behavior is expected as problem scaling results in larger data per node. However, the inverse behavior is observed for CG and LU. Though the actual downtime becomes longer as the task size increases from class B to class C, the absolute downtime (0.27-1.19 seconds) is so small that its effect is subsumed by the variance of overall job execution time (up to 11 seconds for CG and up to 8 seconds for LU), ultimately resulting in an overall decrease in overhead for increasing task sizes.

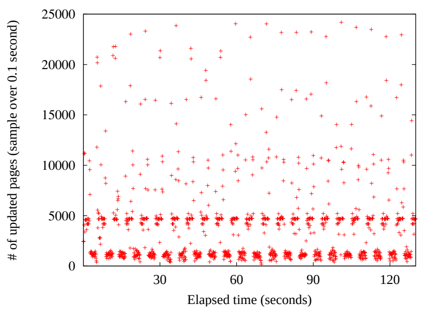
5.4. Effect of Task Scaling

We next examined migration under strong scaling by increasing the number of nodes. Figure 4g depicts the overhead for NPB codes with class C inputs on varying number of nodes (4, 8 or 9 and 16). In most cases, overheads tend to decrease as the number of nodes increases. Yet, BT(Stop&Copy), CG(Live), LU(Stop&Copy) and SP(Live) show no obvious trends. As with problem scaling, this can be attributed to the relatively minor migration downtime, which is effectively subsumed by variances in job execution times. Hence, no measurable effect on task scaling is observed in these cases.

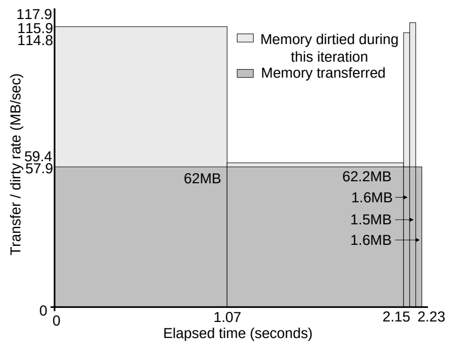
Figure 4h depicts the speedup on 4, 8 or 9 and 16 nodes normalized to the wall-clock time on 4 nodes. The figure also shows the relative speedup of live migration, of frozen migration (stop&copy-only) and without migration. The lightly colored portion of the bars represents the execution time of the benchmarks in the presence of one migration. The aggregate value of light and dark stacked bars presents the execution time without migration. Hence, the dark portions of the bars show the loss in speedup due to migration. The largest loss in speedup is 0.21 with FT on 16 nodes. This can be attributed to FT's relatively large migration overhead (8.5 seconds) compared to its rather short execution time (150 seconds). While the overhead increases proportionately to the memory footprint, the memory footprint is limited by system hardware (total memory size), which also limits the migration overhead. Hence, our results indicate an increasing potential for strong scaling of the benchmarks.



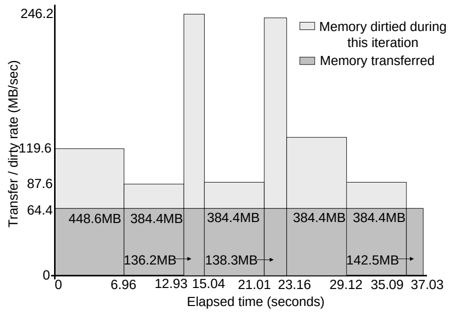
(a) Page Access Pattern of CG C-16



(b) Page Access Pattern of FT C-16



(c) Iterative Live Migration of CG C-16



(d) Iterative Live Migration of FT C-16

Figure 5: Evaluation with NPB for Access Patterns (C-16: Class C on 16 Nodes)

5.5. Page Access Pattern & Iterative Live Migration

Next, page access patterns of the NPB application are analyzed with respect to their effect on live migration, specifically on precopy duration and downtime. Figures 5a and 5b depict the page access pattern of CG and FT with class C inputs on 16 nodes. We sampled the number of memory pages modified per 0.1 second interval. The page size is 4KB on our experimental system. The write frequency to memory pages of CG is lower (∼393/interval) than that of FT (between 1000/interval and 5000/interval). Both codes show a certain regularity in their write patterns, albeit quite an individual one each.

To compare the iterative live migration with the page access pattern, we further evaluated the aggregate amount of the transferred memory and the duration of each iteration of a full memory sweep, as depicted in Figures 5c and 5d. During the first iteration, all non-empty pages are transferred (depicted as dark gray in the figures). In subsequent iterations, only those pages modified since the last iteration (light gray) are transferred. We observed that the write patterns of Figures 5a and 5b are in accord with those depicted in Figures 5c and 5d. For CG, memory pages written in the last two iterations account for 1.6MB corresponding to a write frequency of 393 pages. However, the dirtied memory in the second iteration is only 1.6MB while that in the first iteration is 62.2MB overall, even though the iteration durations are the same. This happens because the first iteration occasionally coincides with the initialization phase of the application. There, higher per-page memory access frequencies (2000-8000 per 0.1 second) persist combined with more frequent cold misses in cache. (This effect was experimentally confirmed but is not explicitly reflected in Figure 5c). For FT, 138MB and 384MB overall of dirtied memory was transferred in an alternating pattern during consecutive iterations (Figure 5d) corresponding to the most significant clusters at 1200 and 4600 page updates in Figure 5b.

These profile patterns of the page modification rate could be archived and subsequently utilized as an empirical factor for future application runs/precopy decisions to choose a more sensitive termination condition (see Section 2). Besides the page access pattern, the communication pattern also affects the precopy decision. For example, if the application is both communication intensive and memory intensive, the opportunity for the precopy operation to complete before an imminent failure is low for high migration overhead and significant network contention. Three main criteria for trading off live and frozen migration and for precopy termination conditions are:

- thresholds, *e.g.*, temperature watermarks in Section 3.1, memory/difference/overhead thresholds in Section 3.2.1;
- available network bandwidth determined by dynamic monitoring; and
- size of the write set. (If the memory dirtying rate is faster than the available network rate, the precopy operation may be ineffective.)

Based on these conditions, a heuristics algorithm can be designed. However, the applications we evaluated are not sufficient to design such an algorithm. In future work, we plan to create and assess applications with varying communication rate and memory access pattern that are more suitable.

5.6. Process-Level Live Migration vs. Xen Virtualization Live Migration

We next provide the performance comparison of our approach to another solution at the OS virtualization layer in the context of proactive FT of MPI applications [42]. The common benchmarks measured with both solutions on the same hardware were NPB BT, CG, LU and SP. For these NPB codes with class C inputs on 16 nodes, the overhead of migrating an entire guest OS ranged between 4-14 seconds under Xen virtualization while the process-level solution caused only around 1 second overhead. The time taken from initiating migration to actual completion on 16 nodes ranged between 14-24 seconds for live migration as opposed to a near-constant cost of 13-14 seconds for frozen migration (stop&copy), both under Xen virtualization. In contrast, our process-level solution only requires 2.6-6.5 seconds for live migration and 1-1.9 seconds for frozen (stop&copy-only) migration. The main difference between the two solutions is that the Xen virtualization solution induced a 13 second minimum overhead to transfer the entire memory image of the inactive guest VM (with a guest memory cap of 1GB) while the process-level solution constrained migration to only the memory of the respective process. Hence, our solution (with 1-6.5 seconds of prior warning to successfully trigger live process migration) significantly outperforms the Xen virtualization solution (with 13-24 seconds of prior warning). One could argue that only a subset of the OS image needs to be migrated, yet the strength of virtualization lies in its transparency, yet it comes at the cost of indiscriminate transfer of the entire virtual memory range.

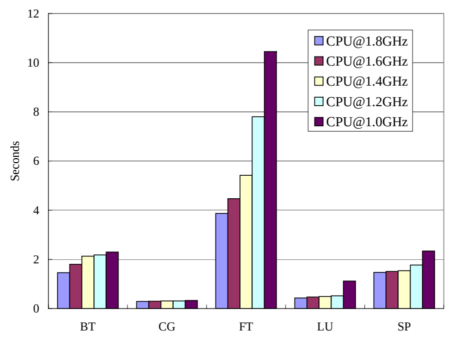
5.7. Proactive FT Complements Reactive FT

We claim that proactive FT complements its reactive sister. This was already empirically shown in Figure 4h where we noted that scalability of live migration depends on the amount of transferred memory. Once local memory is exhausted by an application, the overhead of a single migration will remain constant irrespective of the number of nodes. Of course, the rate of failures in larger systems is bound to increase, but proactive FT supports larger systems while reactive schemes result in increased I/O bandwidth requirements, which can become a bottleneck. This supports our argument that proactive schemes are important since they can complement reactive schemes in lowering checkpoint frequency requirements of the latter.

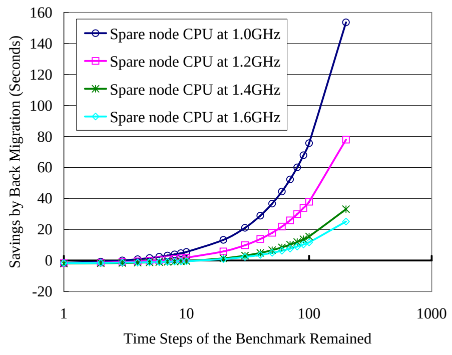
An analytical argument for the complementary nature is given next. The objective here is to assess the ability of proactive FT to successfully counter imminent faults, which subsequently allows reactive schemes to engage in less frequent checkpointing. Let the time interval between checkpoints be T_c, the time to save checkpoint information be T_s, and the mean time between failures (MTBF) be T_f. Then, the optimal checkpoint rate is $T_c = \sqrt{2 \times T_s \times T_f}$ [43]. We also observed that the mean checkpoint time (T_s) for BT, CG, FT, LU and SP with class C inputs on 4, 8 or 9 and 16 nodes is 23 seconds on the same experimental cluster [23]. With a MTBF of 1.25 hours [2], the optimal checkpoint rate T_c is $T_c = \sqrt{2 \times 23 \times (1.25 \times 60 \times 60)} = 455$ seconds. Let us further assume that 70% of failures can be predicted [13] and can thus be avoided by our proactive migration. (Sahoo *et al.* actually reported that up to 70% of failures can be predicted without prior warning; with a prior warning window, the number of proactively tolerated failures could even exceed 70%.) Our solution can then prolong reactive checkpoint intervals to $T_c = \sqrt{2 \times 23 \times (1.25/(1-0.7) \times 60 \times 60)} = 831$ seconds. The challenge with proactive FT then becomes (1) to provide a sufficient number of spare nodes to avoid initial failures by live or frozen migration and (2) to repair faulty nodes in the background such that jobs running over the course of days can reclaim failed nodes for future proactive FT.

5.8. Back Migration

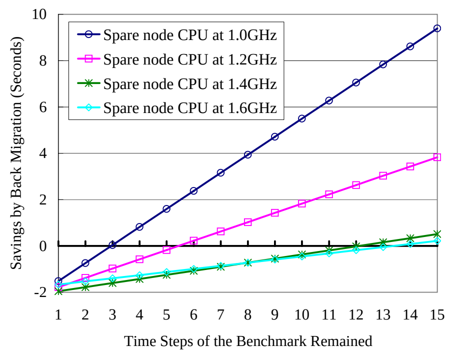
We assessed the back migration approach on the same cluster. Results were obtained for NPB as depicted in Figure 6. Figure 6a shows the downtime fraction of live migration from the original node to a spare for different CPU frequencies at the destination node. For BT, FT, LU and SP, we observe an increase in downtime as the CPU frequency on the spare node decreases, which is expected as slower CPU frequencies result in longer migration. For CG, no obvious increasing behavior is observed since the main computation of CG is matrix multiplication, which is dominated by memory accesses and communication rather than CPU frequency-centric effects. The corresponding overheads of one time step per CPU frequency are depicted in Figure 6d. Figures 6b and 6e depict the savings of back migration for BT and LU. Figures 6c and 6f depict the cross-over region of the previous figures indicating the minimal number of required time steps to benefit from back migration BT and LU are representative



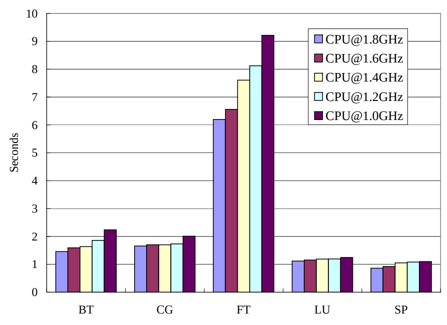
(a) Live Migration Downtime for NPB C-16



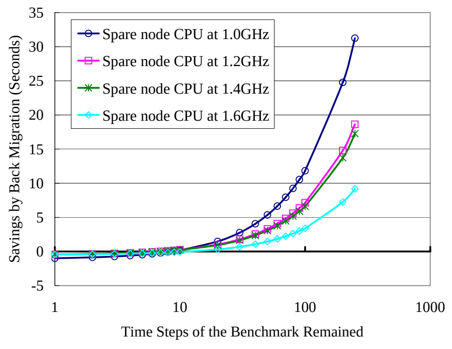
(b) Savings of Back Migration for BT C-16



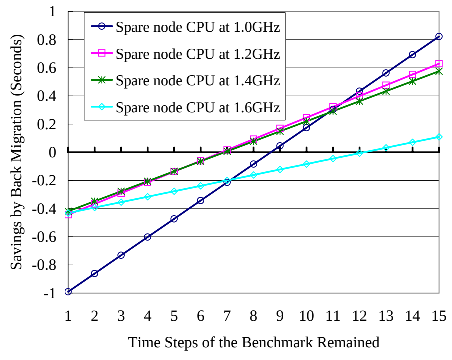
(c) Remaining Time Steps for BT C-16



(d) Overhead of One Time Step for NPB C-16



(e) Savings of Back Migration for LU C-16



(f) Remaining Time Steps for LU C-16

Figure 6: NPB Results for Back Migration (C-16: Class C on 16 Nodes)

for the remaining NPB codes. The results for CG, FT and SP are also measured and assessed but not depicted here since they follow the same trends.

Benefits from the back migration are obtained when

$$R \times (T_d - T_o) - T_m > 0$$

which implies

$$R > T_m/(T_d - T_o)$$

where R is the number of remaining time steps of the benchmark, T_d is the overhead of one time step on the spare/destination node, T_o is the overhead of one time step of the benchmark on the original node, and T_m is the back-migration overhead (assumed to be symmetric to the initial migration overhead to the spare node). T_m plays an important role in the calculation of benefits when the number of remaining time steps (R) is low, which leads to a cross-over point as depicted in Figures 6c and 6f. Benefits due to migration increase as the CPU frequency on the spare node decreases when R is large enough to cover up the effect of T_m.

Table 2 summarizes the time steps required to obtain benefits for BT, CG, FT, LU and SP class C on 16 nodes. The number below the benchmark name is the total

CPU Frequency of Spare Node	**BT (200)**	**CG (75)**	**FT (20)**	**LU (250)**	**SP (400)**
1.6GHz	14 (7%)	7 (9.3%)	13 (65%)	13 (5.2%)	26 (6.5%)
1.4GHz	13 (6.5%)	8 (10.67%)	4 (20%)	7 (2.8%)	9 (2.25%)
1.2GHz	6 (3%)	5 (6.67%)	5 (25%)	7 (2.8%)	9 (2.25%)
1.0GHz	3 (1.5%)	1 (1.33%)	4 (20%)	9 (3.6%)	10 (2.5%)

Table 2: **Minimal Time Steps (Percentage) Remained to Benefit from Back Migration**

number of time steps of the benchmark. The results in the table indicate that benefits from back migration occur when as little as 1.33% and as much as 65% of the MPI job execute time remains to be executed, depending on the benchmark, for an average of 10.19% of outstanding execution time. These results are highly skewed by the short runtime duration of the NPB codes, particularly by FT, which has a comparatively short job execution time and high migration overhead. For larger applications, benefits are projected to occur for even smaller fractions of execution time.

In general, the larger the amount of outstanding execution and the larger the performance difference between nodes, the higher the benefit due to back migration will be. These results illustrate a considerable potential of back migration particularly for large-scale clusters with heterogeneous nodes or multi-hop, non-uniform message routing. This is an aspect without studied investigation, to the best of our knowledge.

6. Related Work

Process-level migration has been studied extensively in the past [44, 45, 46, 47, 48, 49, 19]. In the context of HPC, many MPI implementations have been retrofitted with or design for FT, ranging from automatic methods (checkpoint-based or log-based) [50, 22, 51] to non-automated approaches [52, 53].

System checkpoint-based methods commonly employ a combination of OS support to checkpoint a process image (*e.g., via* the BLCR Linux module [19]) combined with a coordinated checkpoint negotiation using collective communication among MPI tasks. User-level checkpointing, in contrast, relies on runtime library support and may require source preprocessing of an application and typically inflicts lower overhead, yet fails to capture critical system resources, such as file descriptors [54, 55, 56]. Log-based methods exploit messages logging and optionally their temporal ordering, where the latter is required for asynchronous non-coordinated checkpointing. MPICH-V [51] implements three such protocols. It uses Condor's user-level checkpoint library [57]. Non-automatic approaches generally involve explicit invocation of checkpoint routines. In contrast to these methods, ghost process [58] provides a global management service with process virtualization mechanisms supporting process migration within clusters within the Karrighed single-image system. While message logging provides an alternative to coordinated checkpointing, past work on MPI message logging, such as MPICH-V sender/receiver logging, was conducted under the assumption that global synchronization would be too costly at scale. This assumption has proved to be incorrect even for the largest systems today due to scalable custom interconnects that are required for collectives within HPC applications [24, 59].

Recent studies focus on proactive FT. The feasibility of proactive FT has been demonstrated at the job scheduling level [17] and in Adaptive MPI [14, 15, 16] using a combination of (a) object virtualization techniques to migrate tasks, (b) causal message logging [60] and (c) load balancing techniques [61] within the MPI runtime system of Charm++ applications. In contrast to Charm++, our solution is coarser grained as FT is provided at the *process level*, thereby encapsulating most of the process context, including open *file descriptors*, which are beyond the MPI runtime layer. The Charm++ approach only migrates threads within the runtime system and lacks live migration support. Also, back migration provides instant rebalancing instead of adaptive rebalancing over some period of time.

Two facets on proactive FT are intensively studied. First, there are a number of research efforts on failure prediction and management [13, 62, 63, 64]. These papers report high failure prediction accuracy with a prior warning window, which is the premise for our process migration mechanism proposed in this paper. Second, various migration mechanisms are proposed as discussed previously. Furthermore, MPI-Mitten [65], an MPI library between the MPI layer and the application layer, provides proactive fault tolerance to MPI applications. It uses HPCM [66] as a middleware to support user-level heterogeneous process migration. These two facets are integrated in approaches that combine prediction and migration in proactive FT systems and evaluate different FT policies. In [67], the authors provide a generic framework based on a modular architecture allowing the implementation of new proactive fault tolerance policies/mechanisms. An agent oriented framework [68] was developed for grid computing environments with separate agents to monitor individual classes or subclasses of faults and proactively act to avoid or tolerate a fault. Sun *et al.* provide fault-aware systems, such as FARS [69, 70] and FENCE [71], to increase the accuracy of fault prediction and improve system resilience to failures with different fault management mechanisms including process migration. They also model the migration cost, introduce a dynamic scheduling mechanism accordingly and advocate a combined checkpoint/migration system but no live migration [72, 73]. In their paper, Tikotekar *et al.* also present a simulation framework that evaluates different FT mechanisms and policies, including a combination of reactive FT and proactive FT to decrease the number of checkpoints [6], which obtained the best results among all the real and simulated FT mechanisms and policies. These prior works with their fault models, FT mechanisms for fault occurrences and their evaluation simulations, confirm that the process migration is a suitable approach for proactive FT with lower cost than OS virtualization, which reinforces the significance of our solution.

Investigating approaches for transparent real-time performance monitoring of the MPI parallel programs, information on wall-clock time of MPI events collected in prior work [74, 75] with the intent to automatically detect deviations of a process from its expected behavior and subsequently enable real-time scheduling of MPI programs. We, in contrast, contribute a monitoring API that meshes directly with MPI applications to log overheads of time steps, as discussed in Section 3.

Our work enhances LAM/MPI and BLCR [18, 19, 22], which previously was restricted to reactive FT, to a proactive live migration scheme. LAM/MPI+BLCR originally required a complete system restart for roll-back to the last checkpoint upon failure, but a number of approaches have been designed to allow (a) selected checkpoint images to be restarted on new nodes [76], (b) node and head-node failure [77], and (c) a job-pause mechanism that supports migration without restart [23]. Our solution orchestrates the BMC/IPMI health monitoring, the new fundamental utilities provided at BLCR and the extended communication mechanism at LAM/MPI through the decentralized scheduler. The framework is simple and applicable to arbitrary MPI implementations in HPC environment. Furthermore, our approach provides a *live migration mechanism*, which supports continued execution of MPI applications *during* much of the migration time. This solution parallels live migration at the OS virtualization layer [78], which has been studied in the context of proactive FT of MPI applications [42], an approach that supports integrated health-based monitoring and proactive live migration over Xen guests. We contribute *process-level* live migration and demonstrate its superior efficiency to of OS-level virtualization. In HPC, process-level solutions are more widely accepted than OS virtualization, not the least because of potential performance penalties of network virtualization or additional driver development for virtualization-bypass technologies [79, 80].

7. Conclusion

We have contributed a novel *process-level live migration* mechanism with a concrete implementation as an enhancement to the Linux BLCR module and an integration within LAM/MPI. By monitoring the health of each node, a process can be migrated away from nodes subject to imminent failure. We also contributed a mechanism to migrate an MPI task back to the original node once this node is brought back online in a healthy state to eliminate the potential load imbalance or bottlenecks caused by the initially task migrated, which is an unprecedented concept, to the best of our knowledge. We show live migrations to be beneficial to frozen migration due to a lower overall overhead on application wall-clock execution time. This lower overhead is attributed to the asynchronous transfer of a large portion of a migrated process image while the application stays alive by proceeding with its execution. Process-level migration is also shown to be significantly less expensive than migrating entire OS images under Xen virtualization. The resulting proactive approach complements reactive checkpoint/restart schemes by avoiding roll-backs if failures are predicted only seconds before nodes cease to operate. Thus, the optimal number of checkpoints of applications can be nearly cut in half when 70% of failures are addressed proactively.

[1] C.-H. Hsu, W.-C. Feng, A power-aware run-time system for high-performance computing, in: Supercomputing, 2005.

[2] I. Philp, Software failures and the road to a petaflop machine, in: HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of HPCA-11, IEEE Computer Society, 2005.

[3] H. Quinn, P. Graham, Terrestrial-based radiation upsets: A cautionary tale, in: FCCM 05, 2005.

[4] B. Schroeder, E. Pinheiro, W.-D. Weber, Dram errors in the wild: a large-scale field study, SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems.

[5] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: SOSP '03, 2003, pp. 29–43.

[6] A. Tikotekar, G. Vallée, T. Naughton, S. L. Scott, C. Leangsuksun, Evaluation of fault-tolerant policies using simulation, in: IEEE Cluster, 2007.

[7] H. Song, C. Leangsuksun, R. Nassar, Availability modeling and analysis on high performance cluster computing systems., in: ARES, 2006, pp. 305–313.

[8] S. Rani, C. Leangsuksun, A. Tikotekar, V. Rampure, S. Scott, Toward efficient failre detection and recovery in HPC, in: High Availability and Performance Computing Workshop, 2006.

[9] V. Tipparaju, M. Krishnan, B. Palmer, F. Petrini, J. Nieplocha, Towards fault resilient global arrays, in: Parallel computing: architectures, algorithms, and applications, 2008.

[10] J. Varma, C. Wang, F. Mueller, C. Engelmann, S. L. Scott, Scalable, fault-tolerant membership for MPI tasks on hpc systems, in: International Conference on Supercomputing, 2006, pp. 219–228.

[11] T. LeBlanc, R. An, , E. Gabriel, J. Subhlok, Volpexmpi: an mpi library for execution of. parallel applications on volatile nodes, in: European PVM/MPI Users' Group Meeting, 2009, pp. 124–133.

[12] Advanced configuration & power interface, http://www.acpi.info.

[13] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, R. Vilalta, A. Sivasubramaniam, Critical event prediction for proactive management in large-scale computer clusters, in: KDD '03, 2003.

[14] S. Chakravorty, C. Mendes, L. Kale, Proactive fault tolerance in large systems, in: HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of HPCA-11, 2005.

[15] S. Chakravorty, C. Mendes, L. Kale, Proactive fault tolerance in MPI applications via task migration, in: HiPC, 2006.

[16] S. Chakravorty, C. Mendes, L.Kale, A fault tolerance protocol with fast fault recovery, in: IPDPS, 2007.

[17] A. Oliner, R. Sahoo, J. Moreira, M. Gupta, A. Sivasubramaniam, Fault-aware job scheduling for BlueGene/L systems, in: IPDPS, 2004.

[18] J. M. Squyres, A. Lumsdaine, A Component Architecture for LAM/MPI, in: European PVM/MPI Users' Group Meeting, no. 2840 in Lecture Notes in Computer Science, Springer-Verlag, Venice, Italy, 2003, pp. 379–387.

[19] J. Duell, The design and implementation of berkeley lab's linux checkpoint/restart, Tech. rep., Lawrence Berkeley National Laboratory (2000).

[20] J. Hursey, J. M. Squyres, A. Lumsdaine, A checkpoint and restart service specification for Open MPI, Technical report, Indiana University, Computer Science Department (2006).

[21] J. Hursey, J. M. Squyres, T. I. Mattox, A. Lumsdaine, The design and implementation of checkpoint/restart process fault tolerance for Open MPI, in: DPDNS, 2007.

[22] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, E. Roman, The LAM/MPI checkpoint/restart framework: System-initiated checkpointing, in: LACSI, 2003.

[23] C. Wang, F. Mueller, C. Engelmann, S. Scott, A job pause service under LAM/MPI+BLCR for transparent fault tolerance, in: IPDPS, 2007.

[24] N. Adiga, et al., An overview of the BlueGene/L supercomputer, in: Supercomputing, 2002.

[25] A. B. Nagarajan, F. Mueller, Proactive fault tolerance for HPC with Xen virtualization, Tech. Rep. TR 2007-1, Dept. of Computer Science, North Carolina State University (2007).

[26] B. Allen, Monitoring hard disk with smart, Linux Journal.

[27] J. Brevik, D. Nurmi, R. Wolski, Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems, in: IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 190–199.

[28] Top500 supercomputer sites, http://www.top500.org/.

[29] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers, in: Supercomputing, 2005.

[30] Ganglia, http://ganglia.sourceforge.net/.

[31] htop, http://htop.sourceforge.net/.

[32] Performance application programming interface, http://icl.cs.utk.edu/papi/.

[33] M. Pietro, P. RUSSEK, K. Wiatr, Loop Profiling Tool For HPC Code Inspection as An Efficient Methond of FPGA Based Acceleration, Int. J. Appl. Math. Comput. Sci., 1010.

[34] Readable dirty-bits for IA64 linux, https://www.gelato.unsw.edu.au/archives/gelato-technical/2005-November/001080.html.

[35] J. Mehnert-Spahn, E. Feller, M. Schoettner, Incremental checkpointing for grids, in: Linux Symposium, 2009.

[36] M. Vasavada, Innovative schemes to suppport incremental checkpointing, Master's thesis, Dept. of CS, North Carolina State University (Aug. 2010).

[37] mpip: Lightweight, scalable mpi profiling, http://mpip.sourceforge.net/.

[38] I. T. Association, Infiniband, http://www.infinibandta.org/.

[39] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, D. Panda, Rdma-based job migration framework for mpi over infiniband, in: Cluster, 2010.

[40] F. Wong, R. Martin, R. Arpaci-Dusseau, D. Culler, Architectural requirements and scalability of the NAS parallel benchmarks, in: Supercomputing, 1999.

[41] A. Faraj, X. Yuan, Communication characteristics in the nas parallel benchmarks, in: IASTED PDCS, 2002, pp. 724–729.

[42] A. B. Nagarajan, F. Mueller, Proactive fault tolerance for HPC with Xen virtualization, in: ICS, 2007.

[43] J. W. Young, A first order approximation to the optimum checkpoint interval, Commun. ACM 17 (9) (1974) 530–531. doi:http://doi.acm.org/10.1145/361147.361115.

[44] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, Process migration, ACM Computing Surveys (CSUR) 32 (3) (2000) 241–299.

[45] M. L. Powell, B. P. Miller, Process migration in DEMOS/MP, in: Symposium on Operating Systems Principles, 1983.

[46] M. Theimer, K. A. Lantz, D. R. Cheriton, Preemptable remote execution facilities for the V-System., in: SOSP, 1985, pp. 2–12.

[47] E. Jul, H. M. Levy, N. C. Hutchinson, A. P. Black, Fine-grained mobility in the emerald system., ACM Trans. Comput. Syst. 6 (1) (1988) 109–133.

[48] A. Barak, R. Wheeler, MOSIX: An integrated multiprocessor UNIX, in: Proceedings of the Winter 1989 USENIX Conference, USENIX, Berkeley, CA, USA, 1989, pp. 101–112.

[49] F. Douglis, J. K. Ousterhout, Transparent process migration: Design alternatives and the sprite implementation., Softw., Pract. Exper. 21 (8) (1991) 757–785.

[50] G. Stellner, CoCheck: checkpointing and process migration for MPI, in: Proceedings of IPPS '96, 1996.

[51] G. Bosilca, A. Boutellier, F. Cappello, MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes, in: Supercomputing, 2002.

[52] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. Woodall, M. Sukalski, Architecture of LA-MPI, a network-fault-tolerant MPI, in: IPDPS, 2004.

[53] G. E. Fagg, J. J. Dongarra, FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world, in: Euro PVM/MPI User's Group Meeting, Vol. 1908, 2000, pp. 346–353.

[54] M. Litzkow, Remote unix - turning idle workstations into cycle servers, in: Usenix Summer Conference, 1987, pp. 381–384.

[55] J. S. Plank, M. Beck, G. Kingsley, K. Li, Libckpt: Transparent checkpointing under Unix, in: Usenix Winter Technical Conference, 1995, pp. 213–223.

[56] J. Ansel, K. Arya, G. Cooperman, Dmtcp: Transparent checkpointing for cluster computations and the desktop, in: 23rd IEEE International Parallel and Distributed Processing Symposium, 2009.

[57] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, Checkpoint and migration of UNIX processes in the Condor distributed processing system, Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department (April 1997).

[58] G. Vallee, R. Lottiaux, D. Margery, C. Morin, J.-Y. Berthou, Ghost process: a sound basis to implement process duplication, migration and checkpoint/restart in linux clusters, in: ISPDC, 2005.

[59] O. R. N. Laboratory, Resources - national center for computational sciences (nccs), http://info.nccs.gov/resources/jaguar (Jun. 2007).

[60] E. N. Elnozahy, W. Zwaenepoel, Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit, IEEE Trans. Comput. 41 (5) (1992) 526–531.

[61] C. Huang, G. Zheng, L. Kalé, S. Kumar, Performance evaluation of adaptive mpi, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2006, pp. 12–21.

[62] X. Gu, S. Papadimitriou, P. S. Yu, S.-P. Chang, Toward predictive failure management for distributed stream processing systems, in: IEEE ICDCS, 2008.

[63] P. Gujrati, Y. Li, Z. Lan, R. Thakur, J. White, A meta-learning failure predictor for BlueGene/L systems, in: ICPP, 2007.

[64] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, E. J. Weyuker, Performance assurance via software rejuvenation: Monitoring, statistics and algorithms, in: Proc. International Conference on Dependable Systems and Networks, 2006, pp. 435–444.

[65] C. Du, X.-H. Sun, MPI-Mitten: Enabling migration technology in MPI, in: IEEE CCGrid, 2006.

[66] C. Du, X.-H. Sun, K. Chanchio, HPCM: A pre-compiler aided middleware for the mobility of legacy code, in: IEEE Cluster, 2003.

[67] G. Vallée, K. Charoenpornwattana, C. Engelmann, A. Tikotekar, C. B. Leangsuksun, T. Naughton, S. L. Scott, A framework for proactive fault tolerance, in: ARES, 2007, pp. 659–664.

[68] M. T. Huda, H. W. Schmidt, I. D. Peake, An agent oriented proactive fault-tolerant framework for grid computing, in: International Conference on e-Science and Grid Computing, 2005.

[69] Y. Li, P. Gujrati, Z. Lan, X.-H. Sun, Fault-driven re-scheduling for improving system-level fault resilience, in: ICPP, 2007.

[70] Y. Li, Z. Lan, P. Gujrati, X.-H. Sun, Fault-aware runtime strategies for high-performance computing, IEEE Transactions on Parallel and Distributed Systems 20 (2009) 460–473.

[71] X.-H. Sun, Z. Lan, Y. Li, H. Jin, Z. Zheng, Towards a fault-aware computing environment, in: HAPCW, 2008.

[72] C. Du, X.-H. Sun, M. Wu, Dynamic scheduling with process migration, in: IEEE CCGrid, 2007.

[73] Z. Lan, Y. Li, Adaptive fault management of parallel applications for high-performance computing, IEEE Transactions on Computers 57 (2008) 1647–1660.

[74] S. H. Russ, R. Jean-Baptiste, T. S. Kumar, M. Harmon, Transparent real-time monitoring in mpi, in: Springer, 1999.

[75] G. Florez, Z. Liu, S. M. Bridges, A. Skjellum, R. B. Vaughn, Lightweight monitoring of mpi programs in real time, in: Concurr. Comput.: Pract. Exper., 2005.

[76] J. Cao, Y. Li, M. Guo, Process migration for MPI applications based on coordinated checkpoint, in: ICPADS, 2005, pp. 306–312.

[77] A. Tikotekar, C. Leangsuksun, S. L. Scott, On the survivability of standard MPI applications, in: LCI International Conference on Linux Clusters: The HPC Revolution, 2006.

[78] C. Clark, K. Fraser, S. Hand, J. Hansem, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines, in: NSDI, 2005.

[79] A. Menon, A. Cox, W. Zwaenepoel, Optimizing network virtualization in Xen, in: USENIX Conference, 2006.

[80] J. Liu, W. Huang, B. Abali, D. Panda, High performance vmm-bypass I/O in virtual machines, in: USENIX Conference, 2006.

Acknowledgments

This work was supported in part by NSF grants CCR-0237570 (CAREER), CNS-0410203, CCF-0429653, CNS-1058779, DOE DE-FG02-08ER25837 and DOE DE-FG02-05ER25664. The research at ORNL was supported by Office of Advanced Scientific Computing Research and DOE DE-AC05-00OR22725 with UT-Battelle, LLC.