Process Migration for Resilient Applications

TR11-004

Kathleen McGill*
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755
Kathleen.N.McGill@Dartmouth.edu

Stephen Taylor
Thayer School of Engineering
Dartmouth College
Hanover, NH 03755
Stephen.Taylor@Dartmouth.edu

Abstract

The notion of resiliency is concerned with constructing mission-critical distributed applications that are able to operate through a wide variety of failures, errors, and malicious attacks. A number of approaches have been proposed in the literature based on fault tolerance achieved through replication of resources. In general, these approaches provide graceful degradation of performance to the point of failure but do not *guarantee* progress in the presence of multiple cascading and recurrent attacks. Our approach is to dynamically replicate message-passing processes, detect inconsistencies in their behavior, and restore the level of fault tolerance as a computation proceeds. This paper describes novel operating system support for automated, scalable, and transparent migration of message-passing processes. The technology is a fundamental building block upon which to develop resilient distributed applications and has been implemented on a multi-core blade server. To quantify the performance overhead of the technology, we benchmark three distributed application exemplars. Each exemplar is representative of a class of applications solved with a particular decomposition strategy.

Keywords: computational resiliency, fault tolerance, distributed architectures, process migration, resource management

*This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

1 Introduction

Commercial-off-the-shelf (COTS) computer systems have traditionally provided several measures to protect against hardware failures, such as RAID file systems [32] and redundant power supplies [19]. Unfortunately, there has been relatively little success in providing similar levels of fault tolerance to software errors and exceptions. In recent years, computer network attacks have added a new dimension that decreases overall system reliability. A broad variety of technologies have been explored for detecting these attacks using intrusion detection systems [6, 37], file-system integrity checkers [15, 16], rootkit detectors [9, 33], and a host of other technologies. Unfortunately, creative attackers and trusted insiders have continued to undermine confidence in software. These robustness issues are magnified in distributed applications, which provide multiple points of failure and attack.

Our previous attempts to implement resilience resulted in an application programming library called the Scalable Concurrent Programming Library (SCPlib) [20, 21]. Unfortunately, the level of detail involved in programming resilience in applications compounded the complexity of concurrent programming. The research described here explores operating system support for resilience that is automatic, scalable, and transparent to the programmer. This approach to resilience dynamically replicates processes, detects inconsistencies in their behavior, and restores the level of fault tolerance as the computation proceeds [20, 21]. Figure 1(a) illustrates how this strategy is achieved. At the application level, three processes share information using message-passing. The operating system implements a resilient view that replicates each process and organizes communication between the resulting *process groups*. Individual processes within each group are mapped to different computers to ensure that a single failure cannot impact an entire group. Figure 1(b) shows how the process structure responds to attack or failure. An attack is perpetrated against processor 3, causing processes 1 and 2 to fail or to portray communication inconsistencies with other replicas in their group. These failures are detected by communication timeouts and/or message comparison. Detected failures trigger automatic process regeneration; the remaining consistent copies of processes 1 and 2 dynamically regenerate a new replica and migrate it to processors 4 and 1, respectively. As a result, the process structure is reconstituted, and the application continues operation with the same level of assurance.

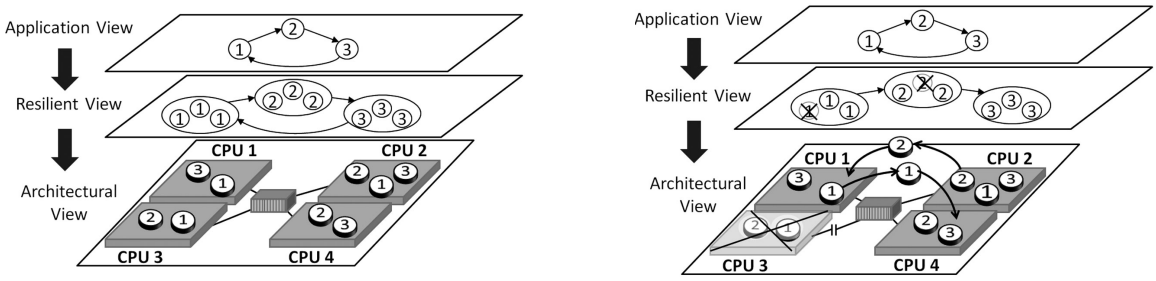

Figure 1: (a) Resilient view of an application running on 4 processors. (b) Dynamic process regeneration after a processor failure.

This approach requires several mechanisms that are not directly available in modern operating systems. Process *replication* is needed to transform single processes into process groups. Process *migration* is required to move a process from one processor to another. As processes move around the network, it is necessary to provide control over where processes are *mapped*. Point-to-point communication between application processes must be replaced by group communication between process groups. It is also desirable to *maintain locality* within groups of replicated processes: This allows message transit delays from earlier messages to be used to predict an upper bound on the delay for failure detection timeouts. Finally, mechanisms to detect process failures and inconsistencies must be available to initiate process regeneration.

This paper describes a process replication and migration technology to support resilience. The tech-

nology has been implemented through Linux loadable kernel modules capable of interrupting a process, packing its state into a message buffer, relocating it to an alternative processor, and restoring it to execution at the new location. A communication module provides a minimal MPI-like message-passing API through kernel TCP functions capable of both point-to-point and multicast communication. The migration module blends prior work in Linux-based migration mechanisms [17, 23, 30, 40, 44] to provide *kernel-initiated* process migration. The unique contribution of this work is the combination of dynamic process replication *and* migration mechanisms in a comprehensive resilient technology. Through cooperation between the communication and migration modules, the kernel can initiate and execute migration of message-passing processes without halting the application or executing global coordination protocols.

To quantify the performance overhead of the technology, we use three distributed application exemplars. Each exemplar is representative of a class of applications solved with a particular decomposition strategy: numerical integration for function decomposition, the Dirichlet problem for domain decomposition, and a distributed LiDAR application [29] that uses an irregular decomposition. The performance impact of the process migration technology is evaluated in the absence replication.

1.1 Related Research

A variety of approaches have emerged to provide *fault tolerance* for distributed applications that rely on the replication of resources. These include checkpoint/restart [11, 13, 35, 43], process migration [2, 7, 28, 40, 41], and process replication [3, 10, 18, 36]. In general, these approaches provide graceful degradation of performance to the point of failure but do not *guarantee* progress in the presence of multiple cascading and recurrent attacks.

Checkpoint/restart systems save the current state of a process to stable storage for recovery and provide the underlying mechanisms to enable process replication and migration. There are many distributed checkpoint/restart systems [1, 4, 13, 22, 35, 43] closely tied with the Message Passing Interface (MPI). These systems emphasize coordination of distributed process checkpoints to achieve a consistent global state of an application through uncoordinated, coordinated, or communication-induced protocols. MPICH-V [4] performs uncoordinated checkpoints using the Condor [22] checkpoint/restart system and message-logging protocols. Starfish [1] modifies the MPI API to enable uncoordinated or coordinated checkpoints. Open MPI [13] integrates the Berkeley Lab Checkpoint Restart (BLCR) system [11] to support alternative coordination strategies. These systems focus on modifications to the MPI library to incorporate global coordination protocols with minimal overhead. In contrast, our technology does not require a global checkpoint: We use process replication to provide fault tolerance, avoiding global coordination protocols altogether.

Process migration is the movement a running process from one host to another [28]. The challenge in migrating message-passing processes is to guarantee message transport during and after migration. Several solutions have been posed at the user-level [24] and kernel-level [2, 5, 40]. Kernel-level solutions have two key advantages that suit the goals of this research: They have kernel privileges to capture all features of the process state, and they operate transparently to applications. This research focuses on kernel-level solutions and builds on prevalent work in Linux-based migration [14, 30, 40, 41].

Kerrighed [23, 40] is a Linux-based cluster operating system that provides a single system image through virtualization of distributed shared memory (DSM) and dynamic communication streams. The DSM simplifies migration because processes can access remote memory on the source host after migration to the destination host. However, this approach creates *residual dependencies*. A residual dependency occurs when a migrated process relies on any resource of the source host after migration. Dependencies are not resilient because failures on the source host can cause the migrated process to fail *after migration*. Our approach differs from Kerrighed in that all residual dependencies are removed after migration.

Cruz [14, 30] is a Linux-based system that migrates entire message-passing applications. It uses a thin virtualization layer to abstract system resources of an application. Cruz preserves transparency by

performing process checkpoints from outside the process context and targeting the lowest level process features (e.g., TCP sockets rather than the MPI interface). Our technology utilizes many of these concepts to preserve transparent migration, but we provide fine grain migration of processes rather than applications.

LAM-MPI [41] is an MPI library that integrates the BLCR system and uses coordinated checkpoints to provide migration for MPI processes. The migration of individual MPI processes faces the distinct challenge of resolving transport for messages that are in-transit during migration. This challenge is complicated by fact that checkpoints are conducted at the kernel level, but the communication state is maintained at the user-level in the MPI interface. The LAM-MPI solution employs user-level hooks in the BLCR system to notify the MPI library of a pending checkpoint. This notification triggers a distinct sequence of actions: computation is suspended, message channels are drained, the process migrates, the message channels are re-established, and the computation resumes. Essentially, this approach disallows in-transit messages by halting the entire application for a single process migration. This global constraint incurs unnecessary overhead in order to retrofit the MPI library for fault tolerance. We remove this constraint by implementing a kernel-level communication module that provides a minimal message-passing API as an alternative to MPI. Message transport is resolved at the kernel level concurrently with process replication and migration.

2 Design and Implementation

Figure 2 shows the software architecture of the technology. It consists of two Linux loadable kernel modules: A communication module and a migration module. These modules are implemented as character devices that provide services to user-level processes through system calls. The technology also utilizes user-level daemon processes and Linux kernel threads. The daemon processes are necessary for tasks that require a user-level address space, such as forking new processes. The message daemon forks processes to comprise distributed applications, and the migration daemon forks processes in which to restore migrating processes. A Linux kernel thread is a process that is spawned by the kernel for a specific function. The TCP servers are kernel threads that receive incoming messages for the communication module. These servers require kernel privileges to access the module functions and memory efficiently.

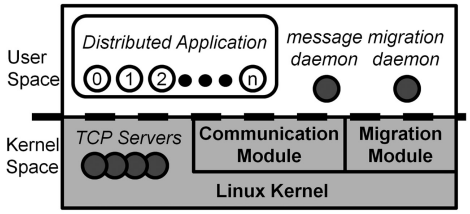

Figure 2: Software architecture of the technology.

The communication module provides a minimal MPI alternative through kernel TCP functions. The API reduces the complexity of the communication state for process migration and has only four basic functions based on blocking communication:

- *msgn(&n)* – sets *n* to be the number of processes in the computation.
- *msgid(&id)* – sets *id* to be the current process identifier within the application.
- *msgsend(dest,buf,size)* – sends a message to *dest* containing the *size* bytes located at *buf*.
- *msgrecv(src,buf,size,&from,&recvd)* – receives a message from *src* into *buf* of *size* bytes in length. The return value, *from*, designates the source of the message, and *recvd* is the number of bytes in the message. If *src* is ANYSRC, then any available message is returned.

In addition, as an artifact of using a character device to implement the module, *msgInitialize()* and *msgFinalize()* calls are used to open and close the device and to provide a file handler for the device throughout the computation. This API can be implemented directly on top of MPI through the appropriate macros. However, the minimalist API is sufficient to support a variety of applications in the scientific community and explicitly disallows polling for messages, a major source of wasted resources. The underlying mechanisms of the API support our resiliency model through the management of process groups, multicast messaging, and failure detection. For example, a blocking *msgrecv()* function is used to provide an environment to detect process failures and trigger process regeneration.

The decision to use our resilient message-passing API in place of MPI was carefully considered. MPI has become the standard for message-passing applications. As a result, significant effort has been made in the research community to retrofit MPI for fault tolerance through distributed checkpoints and process replication or migration. In contrast to these systems, our resiliency model includes both replication and migration. These features would require a major renovation of existing MPI libraries to enforce policies that are not supported by the MPI standard. Thus, we opted to design a low-level implementation to support our preferred resilient message-passing API rather than transform an existing MPI library.

2.1 Communication Module

The communication module provides three primary functions: distributed application initiation, message transport, and migration support. A distributed message-passing program is initiated through the *msgrun* program similar to *mpirun* of the form: *msgrun* -n <n>-r <r><program><args>. The program takes as arguments the number of processes to spawn for the computation (n), the level of redundancy (r), and the executable program name with arguments. The *msgrun* program loads this information into the communication module, and the module sends execution information to remote modules of the cluster. At each host, the communication module signals the message daemon to fork processes for the distributed application.

A key component of application initiation is mapping processes to hosts. A deterministic mapping is used to allow each host to build a consistent process map at startup. Figure 3 shows an example mapping of *n* process groups with resiliency *r* to *m* hosts. Processes are distributed as evenly as possible across the cluster. In order to accommodate resilient process groups, replicas are mapped to maintain locality within process groups without multiple replicas on the same host. The map is constructed by assigning replicas to hosts in ascending order. When there is only one process in a group, processes are mapped round robin. For the purposes of this paper, this mapping creates a worst case scenario in which communication between processes that are adjacent in the process id ordering occurs between hosts. More sophisticated mapping strategies can be found in [12, 24, 25, 26, 27, 42].

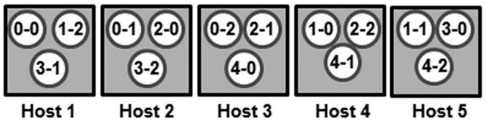

Figure 3: Mapping *n*=5 process groups with resiliency *r*=3 to *m*=5 hosts.

On initiation, an array is allocated at each communication module to store necessary information on application processes. This *process array* stores, for each process, the current mapping, the process id, the process replica number, and a *communication array* to track which hosts have sent messages to the process.

The communication module provides message transport for local and remote application messages. The *msgsend()* call performs an ioctl() on the module. For each call, the module determines whether the destination process is local or remote by referencing the *process array*. Local messages are placed on a message queue in kernel memory. Remote messages are sent via kernel-level TCP sockets. The *msgrecv()*

call uses an ioctl() on the module to search the kernel message queues for the specified message. If the message is in a queue, it is copied to the designated user-space buffer. If the message is not in a queue, the module places the process on a wait queue until the message is received. Both of these calls are extensible to provide multicast messaging in the context of resilient process groups.

Remote application messaging is implemented with persistent sockets created on-demand by the communication module on the message source host. The module manages these sockets in a two-dimensional *socket array* that stores dedicated sockets for each pair of communicating processes. To send an outgoing message, the module first refers to the *socket array*. If a connected socket exists, it is used. Otherwise, a new socket is created and connected for the message. The new socket address is stored in the array for the remainder of the computation, or until a relevant process migration.

On the destination host, concurrent TCP servers manage incoming messages. When the communication module is loaded, it spawns the main TCP server to listen for new connections. For each connection request, the TCP server creates a new socket, accepts the connection, and spawns a new TCP server to receive messages over the socket. This new TCP server handles the socket messages for the duration of the computation unless it receives a message to close prematurely.

The final function of the communication module is to provide support for migration. This includes sending process images that are packaged by the migration module and coordinating message transport for migrated processes. The current proof of concept implementation does not include failure detection. For benchmarking purposes, processes are randomly selected for migration when they enter the *msgrecv()* call. After the migration module has saved and packaged the process image, the communication module executes the migration protocol. First, the local process map is updated to reflect the pending migration. Second, update messages are sent to remote modules stored in the *communication array*. These messages contain the migrating process id and its new host to update remote process maps. No global communication is performed. Third, outgoing sockets for the migrating process are closed. Fourth, the process image is sent to the destination host. Finally, any messages for the migrating process, already in the kernel message queue, are forwarded to the destination host.

In addition to these events, triggered by migration, there are several ongoing functions that the communication module provides to support migration. The process update messages received by TCP servers cause the process map of the local module to be updated and any outgoing sockets connected to the migrating process closed. Throughout the computation, all application messages received by the TCP servers are checked against the process map. If the destination process is no longer local, the message is automatically forwarded to the appropriate host. If the forwarding module is the original host of the process, it also sends a reactive process update message to the message source host. This update message corrects the host process map, so that future communications are direct. This mechanism guarantees message transport for migrating processes; it enables communication to be initiated with the process after migration, and it provides a simple mechanism to address messages that are in-transit during migration.

2.2 Migration Module

The migration module provides migration for message-passing processes by interrupting a process, packing its state into a message buffer, and restoring it to execution at the new location. The process is saved from outside the process context. This tactic may appear to require some manual procedures that would normally be performed by the operating system. However, the alternative is to force processes to save themselves, by modifying user libraries to include special signal handlers or system calls. This allows the operating system to initiate migration transparently without library modifications.

In order to migrate message-passing processes, it is necessary to save critical portions of the process state. The process state includes the memory, kernel, and communication states. The memory state includes the process address space and CPU register contents. The kernel state includes all kernel data structures

that store process information and any system services in use, such as open files. The communication state includes any communication links with other processes and any pending messages.

The migration module mechanisms are tailored to the Linux data structures. Figure 4 shows a simplified diagram of the data structures for each process in the Linux kernel. A unique descriptor called a *task_struct* contains all the necessary information about a process. The *task_struct* includes a pointer to a memory descriptor, *mm_struct *mm* that represents the process address space. The *mm_struct* descriptor includes a pointer to a linked list of virtual memory area structures, *vm_area_struct *mmap*. This list describes all memory areas that are in use by the process. The *mm_struct* also contains a page global directory pointer, **pgd*, which is an index to the first level of the process page tables. These page tables provide virtual to physical memory translation for the process address space. The *task_struct* also stores information on the filesystem (*fs_struct*), open files (*files_struct*), signals (*signals_struct*), and more. The migration module saves a minimal process state for migration, which include the virtual memory, the register contents, the *task_struct*, the *mm_struct*, the *vm_area_struct* list, the *files_struct*, all open files, and some open devices.

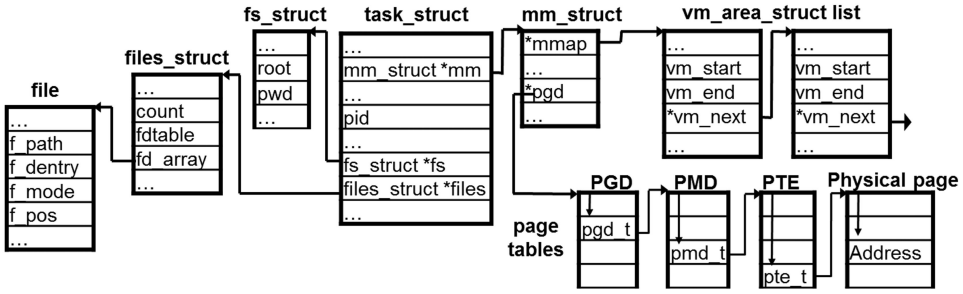

Figure 4: Linux process kernel data structures.

The process save begins with the memory state. The module walks the required memory of the process by traversing the *vm_area_struct* list. However, because the process is out of context, the memory contents cannot be simply copied by the module. For each page of process memory, the page tables are traversed to locate the physical address of the page. If the page is not in memory, it is manually swapped and mapped into kernel memory. The page contents are then copied to the process buffer. To save the communication state of the process, the migration module only saves the open file descriptor associated with the communication module. The remainder of the communication state is addressed by the communication module. Finally, the kernel state is captured in the structures described in Figure 4.

On the destination host, the process is restarted using a custom exec-like function. The migration daemon forks a new process which immediately calls the migration module. The module replaces the forked process image with the buffered process image. The process restore uses existing Linux kernel functions for most of the tasks. The *do_unmap()* and *do_mmap()* functions are used to erase the inherited address space and rebuild the address space from the process image, resolving all virtual to physical memory mappings and process data structures. The CPU context is restored by updating the register contents. Finally, the module reopens any files and devices for the process. Recall that the process migration was initiated during a *msgrecv()* call. The instruction pointer and system call registers are manipulated to force the restored process to repeat the system call upon resumed execution.

The overall migration protocol begins on the source host with the migration module saving the process state and packaging it into a buffer. The communication module completes the migration tasks of the source host by updating the local process map, sending updates to remote modules, closing outgoing sockets for the moving process, sending the process image to the destination host, and forwarding any message for the migrating process already in the message queue. When the process image is received on the destination host, the migration daemon forks a new process, and the migration module restores the process image in

the forked process. The migrated process resumes execution by repeating the original *msgrecv()* call on the destination host. Throughout, when the process update messages are received by remote hosts, the modules update their local process map close and any sockets to the migrating process.

Application messaging may occur simultaneously with the protocol. Messages for the migrating process may arrive at the source host after migration or at the destination host before the process updates are received. Two features of the technology guarantee that these messages eventually reach the destination process: automatic message forwarding and reactive process update messages. The TCP servers will forward messages for migrating processes after migration and send reactive updates to ensure the remote process map is revised. Two scenarios illustrate how the delivery of messages simultaneous with migration is resolved.

Figure 5(a) illustrates a scenario where a message is already in-transit at the beginning of a process migration. Host A is a remote host that sends an application message to the migrating process. Hosts B and C are the source and destination hosts of the migrating process, respectively. The scenario begins when the application message is sent from host A (1). At approximately the same time, the process migration protocol begins on host B. The host B process map is updated, and remote hosts are messaged with the process update (2). Hosts A and C receive the update messages and update their local maps (3). Host B receives the message for the migrating process from host A, but the process is no longer local (4). Host B forwards the application message to host C and sends a reactive process update message to host A (5). Subsequent application messages from host A are sent directly to host C (6). This scenario involves two extra messages: the forwarded application message and the reactive process update message in (5).

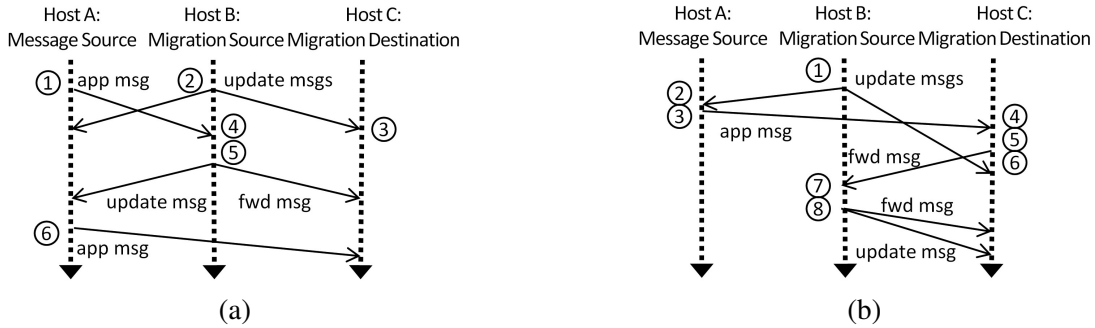

Figure 5: Migration scenarios: (a) message in-transit before process migration and (b) message simultaneous with migration reaches the destination host before the process update message is received.

Figure 5(b) illustrates a second scenario where the migration destination host receives application messages for the migrating process before its process map is updated. This scenario begins with the migration protocol. Host B sends process update messages to hosts A and C (1). Host A receives the process update (2) before sending an application message for the migrating process to host C (3). Host C receives the application message from host A (4) before it receives the process update from host B (6). Host C automatically forwards the application message to host B (5). Host B receives the forwarded message from host C (7) and responds by forwarding the message *back* to host C and sending a reactive process update message (8). This scenario involves three extra messages: the forwarded application messages in (5) and (8) and the reactive process update message in (8). These extra messages in both scenarios represent the costs of guaranteed message delivery without global coordination protocols.

3 Experimental Results

To quantify the performance of migration, three distributed application exemplars are used. Each exemplar is representative of a class of applications solved with a particular strategy: functional, domain, and irregular

decomposition. Numerical integration exhibits the primary characteristics of functional decomposition applications (e.g. Climate Modeling, Floor-plan Optimization) [8] in which a set of components are computed independently. This parameterization decomposes the integral over an interval and sums the integration using a ring messaging pattern. The Dirichlet problem represents domain decomposition applications (e.g. Fluid Dynamics, Computational Chemistry) [38] in which the problem is decomposed over a large, static data structure. It uses an iterative numerical technique that converges to the solution of Laplaces Equation. This parameterization solves the Dirichlet problem using a two-dimensional decomposition in which dependencies are resolved through nearest-neighbor communication. A distributed LiDAR application is typical of irregular problems in which the data structures and algorithms evolve at run-time (e.g. Game playing, Partial Dynamics) [29, 31, 34, 39]. This real-world exemplar uses a sequential kd-tree algorithm to search LiDAR data to construct a digital terrain model. A manager-worker messaging pattern is used to delegate partitions to processes dynamically.

To evaluate the performance of the technology, all three exemplars were tested under a variety of conditions, and the wall-clock times of the computation and migration protocol were measured. The primary metrics used to evaluate performance are the average execution times of the applications, the average overhead of migration, and the average duration of migration. The average overhead of migration is the difference in average execution time of applications with and without migration. The duration of migration is the wall-clock time between the beginning of the process save on the source host and when the restored process exits the migration module to resume execution on the destination host.

These benchmarks were executed on a dedicated Dell PowerEdge M600 Blade Server with 16 hosts. Each host has dual Intel Xeon E5440 2.83GHz processors and 16 GB of memory. The hosts are connected by a 1 Gbps Ethernet network. The operating system is Ubuntu 10.04.01 LTS Linux 2.6.32-26 x86_64. The cluster also has Open MPI v. 1.4.1 for comparison with a standard message-passing system. For each exemplar application, 64 processes were spawned. Three different grid sizes of the Dirichlet problem are included to assess the impact of process memory on the migration technology. The resulting processes have a memory footprint of approximately 9MB, 51MB, and 114MB.

First, the performance of the communication module was evaluated in isolation to ensure that the message-passing implementation does not incur prohibitive overhead. The module performance was compared to Open MPI. Figure 6 shows the average execution times of the exemplars with Open MPI and with the communication module. All average execution times are longer with Open MPI than with the module. The increase in execution time for Open MPI ranges from 11- 64%. These tests were conducted without any effort to optimize either configuration for performance. These results serve as a preliminary verification that the performance of the module is comparable to Open MPI and does not incur unreasonable overhead.

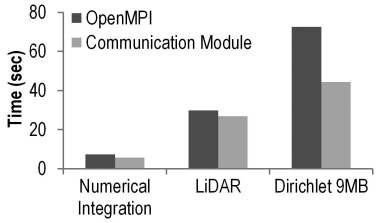

Figure 6: Average execution time of exemplars with OpenMPI and with the communication module.

The second experiment set evaluates the performance impact of process migration on each exemplar. For each computation, one process was selected at random to migrate once. Figure 7(a) shows the average execution times with no migration and with one migration. Figure 7(b) displays the average overhead in execution time due to process migration. The migration overhead ranges from -0.13 to 1.36 sec. For the LiDAR and Dirichlet problems, less than 1% increase in execution time is observed. The LiDAR average

execution time is shorter with migration than it is without migration. This finding is a result of the irregular decomposition of the LiDAR application. The large variance in the execution times of the LiDAR application subsumes the overhead of migration. Migration incurs 3.74% increase in execution time for numerical integration. This result is due to the short total execution time of the numerical integration exemplar.

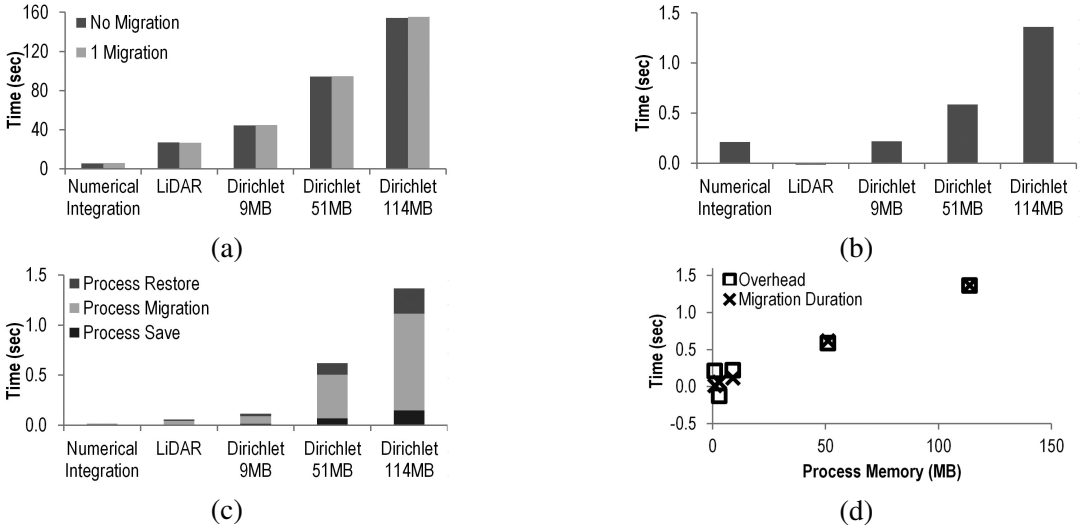

Figure 7: (a) Average execution time of application exemplars with no migration and with a single migration. (b) Average overhead of a single process migration. (c) Average migration duration. (d) Average overhead and migration duration as a function of process memory footprint.

Figure 7(c) shows the migration duration divided into three subtasks: the save, migration, and restore. The migration subtask includes the communication module protocols. The total migration duration ranges from 0.01 to 1.37 seconds. For all exemplars, the majority of the migration time is spent during the migration subtask, and most of this time is spent sending the process buffer. The largest average time that is spent resolving the communication state for the exemplars is 8.9 msec. The overhead in Figure 7(c) is greater than the migration duration depicted in Figure 7(b) for the numerical integration and 9MB Dirichlet problem. This may be because the overhead due to redundant messaging is not explicitly measured. These factors are implicitly included in the overhead and may explain the discrepancy in these metrics.

The relationship between process migration performance and the memory footprint of a process is also clarified by this set of experiments. Figure 7(d) shows the overhead and migration duration as a function of the memory footprint of the migrating process. The migration duration is proportional to the process memory for all applications. The overhead is proportional for the large Dirichlet applications. The latter fact is likely because the migration duration is relatively long for large applications. The overhead may be due to processes waiting for messages from the migrating process itself. From Figure 7(d), the overhead of a single process migration for a large image size can be approximated. The approximate overhead is a fixed 0.08 sec with an addition 0.01 sec/MB.

The final set assesses the impact of multiple process migrations on exemplar performance. In the first tests, concurrent migrations were initiated on multiple hosts. On the specified number of hosts, a single process was selected at random and migrated to the next adjacent host. Figure 8(a) displays the overhead of concurrent migrations from two and four hosts. The numerical integration overhead increases with each additional migration, but the other exemplars show no clear trends. The numerical integration result may be due to the short duration of the application. It may also be related to the ring messaging pattern of the application. The other exemplars suggest that concurrent migrations do not compound overhead.

In the second tests, multiple sequential migrations were conducted on a single host. At five second

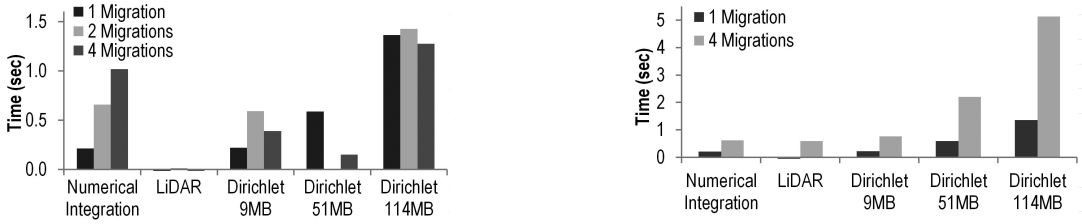

Figure 8: Average overhead of multiple process migration executed (a) concurrently and (b) sequentially.

intervals, one process was selected at random and migrated to a remote host. The remote host was selected in ascending order from the source host. For example, on host 2, processes were migrated to host 3, 4, 5, and 6, in that order. Because the numerical integration has a short duration, migrations were initiated at one second intervals. Figure 8(b) displays the overhead of four sequential migrations for each exemplar. These results show an increase in the overhead for all exemplars, ranging from 0.62 to 5.13 seconds. Much like the relationship shown in Figure 7(d), this overhead is proportional to the process memory footprint. The fixed overhead is estimated as 0.43 sec with an additional 0.04 sec/MB. If these estimates are normalized to a single migration, the fixed overhead is 0.11 sec with an additional 0.01 sec/MB per migration. This finding confirms that sequential migrations have only an additive impact on overhead.

4 Conclusions and Future Work

This paper describes a novel process migration technology for message-passing distributed applications. The technology is based on two kernel modules providing communication and migration. The communication module is a minimal MPI alternative that provides a message-passing API easily adaptable to resilience through multicast communication and resolves message transport for migration of message-passing processes. The migration module packages a process image into a kernel buffer, sends the buffer to a remote machine, and restores the process execution at the new location. This technology achieves migration for message-passing processes automatically and transparently to applications without global communication.

The performance of the technology is quantified using three distributed applications that exemplify functional, domain, and irregular decomposition strategies. For all exemplars, one migration incurs less than 4% overhead in execution time. For multiple recurring migrations, each migration incurs approximately 0.11 sec fixed overhead with an additional 0.01 sec/MB of process memory. This analysis provides a starting point for cost benefit analysis of application resilience. In the future, we would like to explore the tradeoffs in application performance and resilience under different load conditions.

This technology is a central step toward mechanisms that provide application resilience in next generation operating systems. Our approach to resilience is to dynamically replicate processes, detect inconsistencies in their behavior, and restore the level of fault tolerance as the computation proceeds. In a companion effort, we have developed the mechanisms for managing resilient process groups and multicast communications for process replicas transparently. We plan to complete our model by using multicast messaging to automate the detection of process failures as a trigger for process regeneration and migration.

Throughout development, Linux kernel operations for managing data structures and virtual memory have been leveraged extensively. Without similar operations, it would be difficult to implement these solutions in other operating systems. It is clear that this kernel support for resiliency may be simplified considerably. For this reason, we are exploring a new operating system developed from scratch to provide resilience. The design includes data structures, communication protocols, and memory systems to enable dynamic process replication and migration mechanisms that are simple, efficient, and scalable.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 167-176, 1999.

[2] A. Barak and R. Wheeler. MOSIX: An integrated multiprocessor UNIX. In *Proceedings of the Winter 1989 USENIX Conference*, pages 101-112, 1989.

[3] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte. MPI/FTTM: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel. In *Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, 2001.

[4] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, pages 1-18, 2002.

[5] D.R. Cheriton. The V Distributed System, *Communications of the ACM*, 31(3), pages 314-333, 1988.

[6] R. Di Pietro and L.V. Mancini. *Intrusion Detection Systems*. Springer-Verlag, New York, 2008.

[7] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software- Practice and Experience*, 2 (8), pages 757-785, 1991.

[8] I. Foster. *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[9] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 191-206, 2003.

[10] S. Genaud and C. Rattanapoka. P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5(1), pages 27-42, 2007.

[11] P. Hargrove and J. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Proceedings of SciDAC*, June 2006.

[12] A. Heirich and S. Taylor. Load Balancing by Diffusion. In *Proceedings of 24th International Conference on Parallel Programming*, pages 192-202, 1995.

[13] J. Hursey, J.M. Squyres, T.I. Mattox, and A. Lumsdain. The design and implementation of checkpoint/restart process fault tolerance for OpenMPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, 2007.

[14] G. Janakiraman, J. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 260-269, 2005.

[15] J. Kaczmarek and M. Wrobe. Modern approaches to file system integrity checking. In *Proceedings of the 1st International Conference on Information Technology*, 2008.

[16] G.H. Kim and E.H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18-29, 1994.

[17] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 323-336, 2007.

[18] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 124-133, 2009.

[19] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. In *Proceedings of the International Conference on Autonomic Computing*, pages 4-13, 2007.

[20] J. Lee., S.J. Chapin, and S. Taylor. Computational Resiliency. *Journal of Quality and Reliability Engineering International*, 18(3), pages 185-199, 2002.

[21] J. Lee, S. J. Chapin, and S. Taylor. Reliable Heterogeneous Applications. *IEEE Transactions on Reliability, special issue on Quality/Reliability Engineering of Information Systems*, 52(3), pages 330-339, 2003.

[22] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. In *Technical Report CS-TR- 199701346*, University of Wisconsin, Madison, 1997.

[23] R. Lottiaux and C. Morin. Containers: A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66-73, May 2001.

[24] K. McGill and S. Taylor. Comparing swarm algorithms for large scale multi-source localization. In *Proceedings of the 2009 IEEE International Conference on Technologies for Practical Robot Applications*, 2009.

[25] K. McGill and S. Taylor. Comparing swarm algorithms for multi-source localization. In *Proceedings of the 2009 IEEE International Workshop on Safety, Security, and Rescue Robotics*, 2009.

[26] K. McGill and S. Taylor. DIFFUSE algorithm for robotic multi-source localization. In *Proceedings of the 2011 IEEE International Conference on Technologies for Practical Robot Applications*, accepted for publication.

[27] K. McGill and S. Taylor. Robot algorithms for localization of multiple emission sources. *ACM Computing Surveys (CSUR)*, accepted for publication.

[28] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou. Process migration. *ACM Comput.Surv*, 32(3), pages 241-299, 2000.

[29] C. Nichols, S. Taylor, J. Keranen, and G. Schultz. A Concurrent Algorithm for Real-Time Tactical LiDAR. *2011 IEEE Aerospace Conference*, accepted for publication.

[30] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 361-376, 2002.

[31] M.E. Palmer, B. Totty, and S. Taylor. Ray Casting on Shared-Memory Architectures: Efficient Exploitation of the Memory Hierarchy. *IEEE Concurrency*, 6(1), pages 20-36, 1998.

[32] D.A. Patterson, G. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management Data*, pages 109-116, 1988.

[33] N.L. Petroni, T. Fraser, J. Molina, and W.A. Arbaugh. Copilot- a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179-194, 2004.

[34] M. Rieffel, M. Ivanov, S. Shankar, and S. Taylor. Concurrent Simulation of Neutral Flow in the GEC Reference Cell. *Journal of Concurrency: Practice and Experience*, 12(1), pages 1-19, 2000.

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

[36] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D.A. Connors. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 297-306, 2007.

[37] A. Singhal. Intrusion Detection Systems. In *Data Warehousing and Data Mining for Cyber Security*, 31, pages 43-47, 2007.

[38] S. Taylor and J. Wang. Launch Vehicle Simulations using a Concurrent, Implicit Navier-Stokes Solver. *AIAA Journal of Spacecraft and Rockets*, 33(5), pages 601-606, 1996.

[39] S. Taylor, J. Watts, M. Rieffel, and M.E. Palmer. The Concurrent Graph: Basic Technology for Irregular Problems. *IEEE Parallel and Distributed Technology*, 4(2), pages 15-25, 1996.

[40] G. Valle, C. Morin, J. Berthou, I. Dutka Malen, and R. Lottiaux. Process migration based on gobelins distributed shared memory. In *Proceedings of the workshop on Distributed Shared Memory*, pages 325-330, May 2002.

[41] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive Process-Level Live Migration in HPC Environments. In *Proceedings of the 2008 ACM/IEE Conference on Supercomputing*, 2008.

[42] J. Watts and S. Taylor. A Vector-based Strategy for Dynamic Resource Allocation. *Journal of Concurrency: Practice and Experiences*, 1998.

[43] G. Zheng, C. Huang, and L. V. Kale. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), 2006.

[44] H. Zhong and J. Nieh. CRAK: Linux Checkpoint / Restart As a Kernel Module. In *Technical Report CUCS-014-01*, Department of Computer Science, Columbia University, November 2001.