DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems

Joseph F. Ruscio[1] Michael A. Heffner[2] Srinidhi Varadarajan[1]

[1] Computing Systems Research Laboratory
Department of Computer Science
Virginia Tech, VA 24061, USA
{jruscio,srinidhi}@cs.vt.edu

[2] Evergrid, VA 24060, USA
{mike.heffner}@evergrid.com

Abstract

In this paper, we present a new fault tolerance system called DejaVu for transparent and automatic checkpointing, migration, and recovery of parallel and distributed applications. DejaVu provides a transparent parallel checkpointing and recovery mechanism that recovers from any combination of systems failures without any modification to parallel applications or the OS. It uses a new runtime mechanism for transparent incremental checkpointing that captures the least amount of state needed to maintain global consistency and provides a novel communication architecture that enables transparent migration of existing MPI codes, without source-code modifications. Performance results from the production-ready implementation show less than 5% overhead in real-world parallel applications with large memory footprints. [1]

1 Introduction

Enabling the next generation of computational infrastructures, in particular the envisioned national cyberinfrastructure, requires fundamental advances in transparent fault recovery. The large component count inherent in the increasingly popular cluster-based systems increases the instability of the resource as a whole due to the combinatorial dependency of the integrated system on single-component failure rates. Table 1 clearly illustrates that systems with large numbers of cores can expect high failure rates. In such an environment some form of Checkpoint/Restart (CP/R) is the only mechanism that can guarantee large computational jobs will finish. For example in December 2001 LANL ran a complete nuclear explosion simulation on ASCI which consumed 2000 processors for 4 months. Custom application level checkpointing code recorded the job's state every 70 minutes and during the course of that 4 month period the job was restarted over 100 times [12]. Engendering stability in ever-growing, networked, collections of cluster systems needs a software solution that provides reliable access to computing resources through transparent, efficient, and automatic checkpointing and recovery (CPR) mechanisms, a view echoed in the recent emphasis on recovery-oriented computing [16].

We present a new system called DejaVu for transparent and automatic checkpointing, migration and recovery of parallel and distributed applications. DejaVu provides (a) a transparent parallel checkpointing and recovery mechanism that recovers from any combination of systems failures without modification to parallel applications. (b) a novel instrumentation and state capture mechanism that transparently captures application state, (c) novel runtime mechanisms for transparent incremental checkpointing, to efficiently capture the least amount of state required to maintain global consistency, (d) a novel communications architecture that enables transparent migration of existing MPI codes without source-code modifications, and (e) recoverable IO subsystems that can be tailored to specific storage environments.

In this paper, we concentrate on a subset of the capa-

[1]This material is based upon work supported in part by the National Science Foundation(NSF) under Grant No. 0325534. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

1-4244-0910-1/07/$20.00

System	# CPUs	Reliability
ASCI Q	8,192	MTBI: 6.5 Hours, 114 unplanned outages/month. Hardware outage sources: storage, CPU, memory.
ASCI White	8,192	MTBF: 5 hours (2001) and 40 hours (2003). HW outage sources: storage, CPU, 3rd-party HW.
NERSC Seaborg	6,566	MTBI: 14 days. MTTR: 3.3 hours. SW is the main outage source.

Table 1. Reliability and Availability of Large-Scale HPC Systems [18]. (MTBI: Mean Time Between Interruptions, MTBF: Mean Time Between Failures, MTTR: Mean Time To Recovery)

bilities of DejaVu. We present the core algorithm behind DejaVu, proof of global consistency, and its performance on several applications on a cluster with Infiniband interconnect and a shared GPFS file system.

The rest of the paper is organized as follows. Section 2 presents an overview of related work. Section 3 describes the high level architecture and theoretical basis of DejaVu. Section 4 describes the implementation of DejaVu on x86-based Linux systems interconnected with Infiniband. Section 5 presents the results of several experiments used to validate the performance of the CPR framework. Section 6 concludes the paper and presents directions for ongoing work.

2 Related Work

In this section, we present a brief overview of the related work in the area of checkpointing and recovery of distributed systems. The survey presented in [9] provides a much more comprehensive evaluation of the large volume of work in the area of rollback-recovery protocols for distributed systems. From the perspective of our work, rollback recovery protocols can broadly be classified along two dimensions — coordination model and implementation level.

2.1 Coordination Model

Checkpoint coordination model describes how a distributed CP/R system orchestrates the individual checkpoints taken by each process to create a consistent aggregate global checkpoint. CP/R system's coordination models are classified as either uncoordinated or coordinated. Uncoordinated checkpointing methods do not synchronize the checkpoints of individual processes, rather each process schedules and takes its checkpoints independently. Dependency information is tracked to determine the set of individual checkpoints that comprise a globally consistent state. Each process must maintain multiple (if not all) sequential checkpoints, as a dependency will most likely exist from some process's most recent checkpoint to an older checkpoint of another process. Uncoordinated checkpointing is vulnerable to the *domino effect* [9] that can cause the system to roll all the way back to its initial state. Uncoordinated checkpointing schemes were conceived when network communication operations were more expensive then storage I/O (the inverse is now true), and have all but vanished in modern systems.

Coordinated checkpointing methods synchronize the individual checkpoints of each process to ensure that their aggregate is a globally consistent state of the system. Coordinated checkpointing is not susceptible to the domino effect and only requires the most recently recorded checkpoint of each process. This characteristic places an upper bound of one checkpoint interval (period of time between checkpoints) on the amount of compute time lost to a failure.

Coordinated checkpointing is the method used by most if not all modern checkpointing systems and the most commonly implemented algorithm is Distributed Snapshots [4]. This is a global state detection mechanism that achieves coordination through the use of *marker* messages. It relies on a fundamental assumption that the communication channels of the distributed system are reliable, FIFO (First In First Out) queues that guarantee all messages sent by one process to another are received in-order and without error. The marker messages flush each communication channel (and thereby the network) of all messages thereby restricting the state of the distributed computation that must be recorded to that of the individual processes. In practice, some implementations will go a step further and also tear down all the flushed connection endpoints prior to checkpointing, and recreate them after the checkpoint is finished, to enable migration [19].

DejaVu is a coordinated checkpointing system, but unlike Distributed Snapshots it uses a novel runtime mechanism described in §3.2 to capture the state of communication channels as part of the checkpoint and does not incur the overhead associated with flushing the network. DejaVu also virtualizes all communication channel endpoints (e.g. sockets) used by an application to enable migration without tearing down and restarting channels.

2.2 Implementation Level

The implementation level of a distributed CP/R system refers to how it integrates with applications, communication middleware, and the OS. This integration can be classified as application level, user level, and system level.

Due to the lack of a standardized, portable, transparent checkpointing solution, the most common approach to checkpointing today is the application level. Application level checkpointing does not require any modifications to the OS and when implemented properly, minimizes the amount of state stored for each checkpoint. Some application developers add snapshot ability to their codes that periodically dumps the state of the computation in a format that is reloaded in the case of a failure. Other developers structure their applications such that users may split their simulations into intervals that fall underneath the Mean Time Between Failures (MTBF) of the deployment environment and *pipeline* the output of one run into the next. The first method is error-prone, and oftentimes inflexible. It requires application developers to design and implement behaviors that fall outside their domain expertise and the precise interface for CP/R operations is non-standard. Both methods place an undesirable burden on the application end-user to learn and then operate the CP/R facilities.

User level frameworks are an attempt to provide fault-tolerance to applications while requiring minimal effort on the part of application developers and end-users. Sometimes they do not require any modifications to the OS. To a varying extent they are transparent to the end-user and applications. The least transparent methods are checkpoint libraries that provide a checkpoint API that application developers can insert into their codes at the appropriate places. These require developers to manually instrument their code. Compiler-based methods [2] run application source code through a special translator that attempts to infer the optimal points in execution to take checkpoints. These are able to minimize the amount of state recorded in a checkpoint, but require access to application code and do not permit truly asynchronous checkpointing, as checkpoints are taken at specific locations in execution.

Communication middleware implementations are becoming more prevalent amongst user-level CP/R frameworks and embed distributed checkpoint behaviors in an MPI library. These implementations [1] [11] [19] typically focus on the synchronization of checkpoints and either rely on integration with a single process checkpointer (potentially in the OS) to complete the framework or perform in-memory checkpointing for applications with small memory footprint. These are transparent in that they require no access to application codes, but they tie deployment environments to a specific implementation of a messaging-passing API.

DejaVu is a transparent user level CP/R system. What differentiates DejaVu from the other user-level frameworks described above is the higher degree of transparency that it achieves. It requires no access to application codes and its complete BSD socket virtualization supports any MPI implementation built on socket communication. For the high-performance Infiniband interconnect (where native performance is desired) we provide a version of MVAPICH [14] modified at the ADI layer to implement the DejaVu algorithm.

System level checkpoint implementations are housed in the operating system and have the advantage of being completely transparent to the user application. For instance, the AIX operating system from IBM provides CP/R capabilities. There are several implementations [7] [10] for the Linux operating system kernel. These systems are typically single process checkpointers and must be paired with a user-level mechanism for checkpoint coordination. Portability is a major issue for system-level checkpointers. Even on a single OS platform, any revision (even minor ones) to the OS potentially requires porting work.

3 Architecture

3.1 Transparent User-Level Framework

DejaVu is a transparent user-level CPR framework. We believe that transparent user-level frameworks are more desirable then either application-level solutions or system-level frameworks. Application-level solutions by nature are non-standard and for reasons described in §2.2 often ad-hoc and error-prone. At the opposite end of the spectrum, system-level frameworks offer extreme standardization but are not portable across different operating systems, and oftentimes even across different versions or distributions of the same operating system. As described in §2.2 the many other user-level frameworks are the combination of a specialized MPI implementation and a single process checkpointer. These frameworks are application transparent and operate at the communication layer middleware to implement the Distributed Snapshots algorithm. Due to a novel global state detection mechanism combined with BSD socket virtualization, DejaVu is able to virtualize at the OS interface, making it transparent to both applications and any sockets-based communication middleware. This feature allows users to provide their jobs with standardized fault-tolerance. Figure 1 depicts the integration of the DejaVu framework with a typical distributed application. DejaVu intercepts system library calls made by either the application or any middleware libraries it is linked against, requiring no modification to legacy binaries. Consistency is achieved through a transparent online logging protocol, which relaxes the tight synchronization

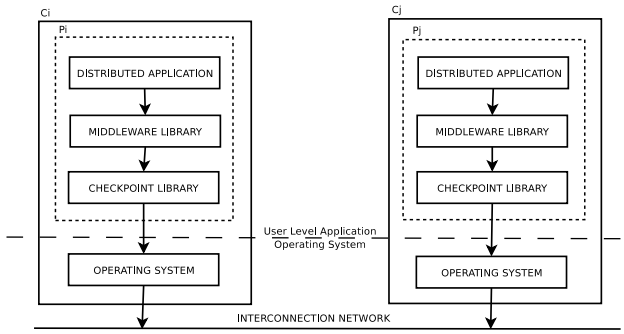

Figure 1. DejaVu virtualizes the application layer by intercepting system calls between the application/middleware and the operating system.

requirements of Distributed Snapshots. We also provide a customized version of the MVAPICH library which implements the DejaVu algorithm to enable fault tolerance of MPI codes running on Infiniband networks while maintaining native performance.

3.2 Online Logging Protocol

DejaVu utilizes an online logging protocol to ensure that the state of a communication channel and the two processes at either end remain consistent during a checkpoint. Its design springs from a single fundamental observation that is completely opposed to the assumption made by distributed snapshots based systems, that all communication channels are reliable FIFO queues. In contrast, DejaVu assumes that all communication channels provided by the OS are *inherently unreliable*, providing no guarantee of delivery or correctness of order. This is a strange assumption to make in light of the last decade's proliferation of high performance system area interconnects. The fundamental observation is that in such an environment, it is impossible to determine whether the loss, out-of-order delivery, or duplication of a message is due to a failure of the sender's OS, the failure of the interconnection network, the failure of the OS on the node at the remote end of the communication channel, *the failure of the process executing on that remote end, or inconsistencies that occurred during the checkpoint itself.* The implications of this observation are profound in that the problem of coordinating tightly coupled distributed systems to ensure consistency reduces to that of reliable transmission over unreliable channels. Therefore DejaVu does not explicitly checkpoint state within the OS nor in-flight over the communication fabric. Nor does DejaVu engage in coordinated "flushing" of messages out of the OS and communication fabric before checkpointing, as do many conventional systems. Instead, the online logging implemented by DejaVu's user-level library *masks any loss/corruption of state within the operating system and the interconnection network during the checkpoint operation as a message loss that would be dealt with in the normal flow of communication over an unreliable channel.*

All communication operations invoked by either the application or middleware are intercepted by the checkpoint library. These requests are carried out utilizing the communication primitives provided by the underlying OS. All communication channels are assumed to be unreliable, and DejaVu ensures correct delivery of messages through use of online logging protocol. In the following description all actions taken by a sending process P_s and a receiving process P_r are undertaken by the DejaVu framework on behalf of the application. P_s "commits" any message m_i (where i is the sequence number) it transmits to a local log. Upon receipt of m_i, P_r replies with an acknowledgment a_i to inform P_s of successful delivery. This action, taken by P_r, "uncommits" the message from P_s's log. A unique, monotonically increasing sequence number i is associated with each message so that messages received in the wrong sequence may be re-ordered properly by P_r. The sequence numbers allow a receiver to detect that a message was lost (a gap in sequence numbers of messages received), as well as the receipt of duplicate messages. Duplicate messages are discarded (as they were already received and processed), while lost messages/acknowledgments are handled by a timeout mechanism. Messages that are in the sender log are known as "outstanding" messages. In order to permit the use of a finite buffer for the log, a limit is placed on the number of outstanding messages that are allowed at any one time. If this limit is reached, P_s ceases sending messages until the oldest outstanding message is acknowledged.

A major deterrent to aggregating uncoordinated local checkpoints from P_s and P_r at either end of a communication channel into a globally consistent checkpoint has been the inherent difficulty in preventing inconsistencies in the communication channel state from occurring upon restart. The following inconsistencies that can arise due to a checkpoint taken while P_s is sending a message m_i to P_r are:

- m_i was sent prior to the checkpoint taken by P_s, but received after the checkpoint taken by P_r. Upon restart P_s has sent a message that will never be received.
- m_i was sent after the checkpoint taken by P_s but received prior to the checkpoint taken by P_r. Upon restart P_r has received a message that was never sent.
- As DejaVu also introduces a secondary acknowledgment message a_i from P_r to P_s there is a third possible inconsistency when a_i is sent prior to the checkpoint

taken by P_r but received after the checkpoint taken by P_s. Upon restart P_s has sent a message that will never be acked.

In the first failure mode, P_s is restarted in a state after having sent m_i and P_r is restarted in a state prior to the receipt of m_i, and as such m_i will never be received by P_r. The online logging protocol prevents this inconsistency by detecting the loss of m_i. As an outstanding message in the log, the timer associated with m_i will expire and m_i will be retransmitted. P_r now receives the second transmission of m_i and uncommits it from the log by replying with a_i.

In the second failure mode, P_s is restarted in a state prior to m_i having been sent and P_r is restarted in a state where m_i has already been received, creating an orphan message. The online logging protocol prevents this inconsistency by detecting the successful delivery of m_i. P_s undertakes what it believes to be the first transmission of m_i, and P_r will recognize from the sequence number that it is a duplicate message. The message will be acknowledged by P_r and discarded.

The third mode of failure is prevented by a combination of the previous two mechanisms. As a_i is never received, P_s will timeout and retransmit m_i. When P_r receives m_i it will be detected as a duplicate and a_i will be retransmitted.

The mechanics of the online logging protocol also provide DejaVu with the the ability to transparently *migrate* processes at restart time to any host in the distributed system. Typically migration is non-trivial since the applications and/or communication middleware contain state that refers to OS communication channels tied to the network location of a specific host. Since DejaVu virtualizes all communication channels to implement the online logging protocol, all that is required to enable migration is to also virtualize network addresses, meaning that all application and middleware requests for OS communication interfaces are satisfied with addresses generated by DejaVu. Internally, these addresses are mapped to the actual network addresses of the compute nodes. Upon a restart the mapping of virtual to real addresses is regenerated to match the new host locations.

3.3 Global Coordination of Checkpoints

The online logging protocol described above clearly ensures the consistent state of a communication channel shared by two processes when restarting from a checkpoint. DejaVu augments this with a mechanism to coordinate the checkpoints taken by all of the individual processes in a distributed job to ensure that the aggregate result is globally consistent. This mechanism works as follows. At some point during a job's execution (most likely periodically) a checkpoint is invoked by sending a signal to any one process in the distributed system. This process, called the P_{root} then initiates a global checkpoint operation by broadcasting a checkpoint command to all processes taking part in the distributed computation. P_{root} may be any one of the processes taking part in the computation or a "third-party" process such as a scheduling entity. The broadcast command contains an identifier called an epoch identifier, which is a monotonically increasing number that identifies a checkpoint interval. Upon receiving the broadcast each process P_i enters a "freeze" period. The freeze does not imply a halt in local execution, just that all inter-process communication is suspended and queued. The online logging protocol ensures that any messages lost/discarded during the freeze are recoverable. After freezing P_i commits the local state that comprises its checkpoint. P_i then concludes the checkpoint operation by entering a "barrier". The barrier is implemented by each P_i notifying P_{root} out-of-band that P_i's state has been committed. The barrier operation is relatively quick, since all processes of the distributed system involved in the computation receive their checkpoint message separated by no more than the network latency, and hence enter the barrier temporally close to each other. When P_{root} receives commit confirmation from each P_i included in the computation, it transmits an out-of-band broadcast declaring the checkpoint operation successful. Upon receipt of this broadcast each process P_i exits the barrier, "unfreezes" and resumes inter-process communication. In contrast to prior systems DejaVu's entire checkpoint phase is only loosely coordinated by a terminating barrier.

3.4 Proof of Global Consistency

Consider a distributed computation comprised of n processes. A global state of the system can be abstractly defined as the union of the individual state of each process P_i and the state of the network. For the purposes of this proof we will assume that the distributed system is a message passing system, but it should be noted that the proof is also correct for shared memory systems where communication between processes is facilitated through the modification of shared memory locations. We denote the local state of P_i as S_i^{proc} and consider it to contain all user-level state such as stack contents, heap contents, register contents, global data (including the data segments of shared libraries), the sender logs associated with open communication channels and all the meta-data pertaining to virtualized operating system primitives. The set of individual process states is defined as

$$S^{proc} = \bigcup_{i=1}^{n} S_i^{proc} \quad (1)$$

We denote the state of a communication channel from P_i to P_j as S_{ij}^{comm} and consider it to contain all state that exists end-to-end between the two processes including any

state in OS kernel buffers, NIC buffers, or in transit on the wire. S_{ij}^{comm} can therefore be defined as a set of messages where each message has been sent by either P_i or P_j and not yet received by the corresponding process. If we define m_i to be a message sent from P_i to P_j and $sn_{i_{r-1}}$ to be the sequence number of the last m_i received by P_j and sn_{i_s} to be the sequence number of the last m_i sent by P_i then the set of messages sent from P_i to P_j but not received is defined as

$$M_i = \{m_{i_k} | k \in \{sn_{i_r}, \ldots, sn_{i_s}\}\} \tag{2}$$

for messages sent from P_j to P_i we also define

$$M_j = \{m_{j_k} | k \in \{sn_{j_r}, \ldots, sn_{j_s}\}\} \tag{3}$$

The state of a communication channel between P_i and P_j is thus defined as

$$S_{ij}^{comm} = M_i \bigcup M_j \tag{4}$$

The state of the network can be considered as the set of the state of all communication channels defined as

$$S^{comm} = \bigcup_{i=1,j=1,i\neq j}^{n} S_{ij}^{comm} \tag{5}$$

Then for distributed computation running on such a system the global distributed state S^{global} can be defined as

$$S^{global} = S^{proc} \bigcup S^{comm} \tag{6}$$

Due to the use of the online logging protocol, every message belonging to S_{ij}^{comm} has been committed to the sender at either P_i or P_j or both. Recall that the contents of the online log associated with sender P_i is contained in S_i^{proc}. This means that the state of the communication channel S_{ij}^{comm} is completely contained in the individual process states S_i^{proc} and S_j^{proc}. The global distributed state then reduces to

$$S^{global} = S^{proc} \tag{7}$$

The role of each process P_i in the global checkpoint operation has been reduced to executing a local checkpoint operation to record S_i^{proc}. In order to ensure the consistency of S^{global} no state S_i^{proc} may change during the local checkpoint operation. More specifically upon entering the global checkpoint operation no process P_i may change (a) its local state and (b) the state of any other process P_j until the global checkpoint operation is finished. The only self-inflicted cause of local state change is local computation. Likewise the only manner for P_i to change the state of P_j is to send a message.

Given these criteria, recall that upon entering the global checkpoint process P_i stops local computation and enters a "freeze period" during which all interprocess communication is suspended. P_i then executes the local checkpoint operation and exits the global checkpoint operation by entering the loosely synchronized out-of-band barrier operation. At no point during its part in the global checkpoint operation does P_i alter its state or send a message to any process P_j that would alter S_j. While in the loosely synchronized barrier operation, P_i will refrain from any interprocess communication. This ensures that P_i does not alter the state of any process P_j that may still be taking part in the global checkpoint operation. Only after every process enters the barrier does P_i resume interprocess communication, thus satisfying the requirements for global consistency of checkpoints taken by DejaVu.

4 Implementation

The current implementation of DejaVu is in the form of a dynamically linked library, libdv.so. This library exports all the POSIX (and non-POSIX) symbols that correspond to library and system calls that DejaVu needs to virtualize. At runtime the LD_PRELOAD environment variable is used to instruct the linker to load the DejaVu library into the application. The linker resolves all application and other middleware library system calls to the exported symbols of DejaVu. During execution either a checkpoint or preemption (checkpoint followed by an exit) operation can be instantiated by signaling the DejaVu linked processes with predefined POSIX operating system signals. These operations can therefore be controlled with something as simple as a parallel shell and the UNIX *killall* command. All configuration of DejaVu's runtime behavior is accomplished via environment variables.

Our current version of DejaVu includes several checkpointing optimizations and features that exceed the scope of this paper. The aggregate of the following list differentiate it from any other checkpoint/restart framework we are aware of.

- *Permits completely asynchronous checkpoint requests.* There are no critical sections or operations during which checkpoints are not allowed. Many application/user level frameworks only permit checkpoints at certain points in execution.
- *Completely decoupled from any specific queuing system or framework.* All interaction with DejaVu takes place through environment variables and signals, permitting easy integration with pre-existing queuing environments.
- *Application and OS transparent.* DejaVu requires no modification of application binaries or operating system kernels. This renders it extremely portable to any platform running a POSIX compliant operating system.

- *Support for anonymous mmap().* DejaVu permits and tracks all anonymous mmap()'s including those made by an application to some fixed location. This increases complexity as dynamically allocated memory can fall almost anywhere in the entire virtual memory space as opposed to just the brk region, but it is absolutely required of a production-capable system.
- *MPI over Infiniband Support.* The current implementation intercepts MPI operations at the ADI layer in MPICH using a modified version of Ohio State University's MVAPICH stack [14]. All results in §5 were from MPI jobs running over an Infiniband network.
- *Support for rolling back file I/O operations.* DejaVu includes a filesystem rollback module that intercepts all UNIX VFS filesystem calls including both buffered and unbuffered operations. As part of the restart protocol it rolls back all changes made to the filesystem since the last checkpoint took place. It uses a copy-on-write method to track inode mutations and also tracks changes to the filesystem hierarchy.
- *Incremental checkpointing.* DejaVu utilizes OS memory protection mechanisms to detect the minimal amount of changed state between checkpoints, modeled after the scheme in [17].
- *Asynchronous concurrent checkpointing.* DejaVu utilizes an optimized low-latency concurrent checkpointing system somewhat similar to that put forth in [13]. DejaVu uses an asynchronous storage thread that overlays computation with checkpoint storage thereby minimizing the impact on application efficiency. Several environment variables can be set to tune the parameters of the disk I/O operations to the deployment environment. DejaVu performs all disk access with standard POSIX system calls and does not require raw disk access.
- *Fixed upper bound on stable storage requirement.* Incremental checkpoints taken by DejaVu are patched into the checkpoint file resulting from the previous checkpoint. Thus DejaVu never stores more then one copy of any given virtual memory page. This allows us to place an upper bound on storage requirements that is defined by the memory footprint of the application. This is in contrast to systems that save each incremental checkpoint independently, resulting in multiple copies of the same page on disk and unbounded storage requirements.
- *Support for forked processes.* DejaVu will record when new processes are created with fork() and enable checkpoint/restart for them too. At checkpoint time the entire process tree will be checkpointed and restarted correctly. IPC mechanisms using file descriptors that are shared between forked processes (e.g., pipes and socket pairs) are correctly restored during restart by reopening them before the child processes are respawned. There is additional on-going work to support all IPC mechanisms including: semaphores and shared memory. Fork support allows checkpoint/restart functionality for commercial HPC applications that typically spawn additional processes.
- *Configurable page size for checkpoint performance.* To improve the performance of checkpointing large memory applications to disk, DejaVu can increase the page size it will fault application memory on. Larger values improve checkpoint performance when checkpointing to file systems with larger optimal file block size.
- *Migration capable.* As described in §3.2 DejaVu virtualizes all location specific information such as open sockets so that a distributed job may be restarted on any set of nodes in the system that is equivalent in size to the set on which it was originally executing.

5 Evaluation

To evaluate the performance overhead of application virtualization and checkpointing we performed tests with High Performance Linpack (HPL), Chombo, and Sander. HPL is an implementation of the LINPACK benchmark that "generates, solves, checks and times the solution process of a random dense linear system of equations on distributed-memory computers." [6] The additional error residual check at the end of each HPL run certifies that the computed result is correct, verifying that DejaVu has correctly restored all memory and register locations. Choosing to evaluate our performance against HPL allowed us to test a code with high temporal and spatial locality.

Chombo is a set of C++ classes designed to facilitate development of applications that use block-structured Adaptive Mesh Refinement (AMR) [5]. Our tests used the cell-centered Poisson solver sample application distributed with Chombo. This application rigorously tested DejaVuś memory allocation virtualization. Sander is a FORTRAN application that is released as part of the AMBER [3] package of molecular simulation programs. Sander writes continuous streams of atomic coordinates and velocities to output files during execution. Sander was used to validate DejaVu's ability to transparently virtualize both FORTRAN applications as well as applications that perform large amounts of file I/O.

The majority of our experimental evaluations were done on the National Energy Research Scientific Computing Center's Dual 2.2GHz AMD64 Opteron cluster running the Linux operating system. This cluster had four compute nodes and one head node. Each node had 6GB of PC3200 DDR memory. The nodes were interconnected with high-speed 4X Infiniband and a secondary Gigabit Ethernet control fabric. Each node was also connected using IP over IB

to a shared IBM General Parallel File System (GPFS) with 300GB of accessible disk space and 200 MB/sec of per-node storage bandwidth. The cluster was deployed with the PBS batch queue system.

We also used an eight node Dual 2.0GHz AMD64 Opteron Linux cluster for our scale up test to sixteen processors. Each node had 4GB of PC2700 DDR memory and they were interconnected with high-speed 4X Infiniband with a secondary Gigabit Ethernet control fabric. The cluster did not have a parallel file system so checkpoints were dumped to the local hard drives in each node.

Checkpoint overhead was measured by taking a single "full checkpoint" during the run of a multi-process job. Measuring the checkpoint overhead of the first checkpoint in an incremental checkpointing system will typically provide an upper bound for the worst checkpoint case. With incremental checkpointing, only pages modified after the first checkpoint must be committed to disk in the next checkpoint. In the following experiments, the runtime of any single run was under an hour of wall-clock time and the problem size was selected to fill the maximum available memory on each node. We assume that a reasonable periodic checkpoint interval for a fault-tolerant system would be set to a period greater than or equal to an hour. Therefore, by measuring the overhead of a single checkpoint within the first hour of runtime using the maximum memory resources, we can assume that our overhead will remain consistent or decrease if the job were to run longer.

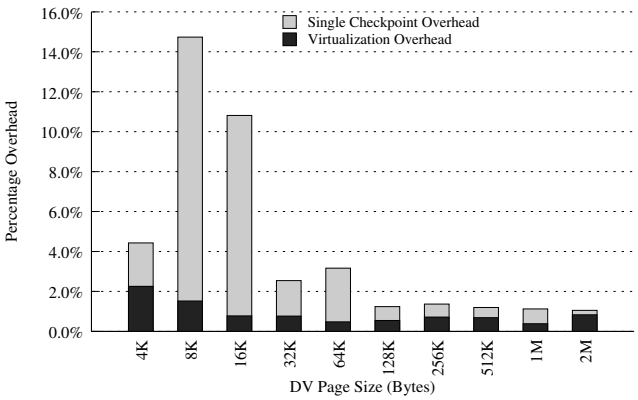

Figure 2. Comparison of DejaVu overhead for HPL vs. normal execution without DejaVu

The most important overhead that a transparent checkpoint recovery system must reduce is the overhead from virtualization. The virtualization overhead — caused by system call interception and state capture — is a constant cost over the entire execution of the application. Therefore, unlike checkpoint overhead, which can be reduced by changing the checkpoint interval, virtualization imposes a continuous overhead. Figures 2, 3, and 4 show the combination of virtualization overhead and single checkpoint overhead for HPL, Chombo, and Sander respectively. Virtualization overhead is measured by comparing runtimes of the application with DejaVu loaded and the baseline without DejaVu loaded. This overhead shows the cost required for doing message logging on high-speed IB networks. Checkpoint overhead was measured by comparing runtimes with and without a single checkpoint taken. In figure 2 it is clear that for HPL the virtualization overhead is less than 1% for page sizes greater than 64KB. In figure 3 the overhead for Chombo is less than 5%. Memory thrashing exhibited by Chombo leads to slightly more overhead than HPL. Figure 4 shows that the overhead was mainly from the virtualization which is not surprising given that Sander is CPU bound and had a small memory footprint. For all the applications the combined overhead at the larger DejaVu page sizes is less

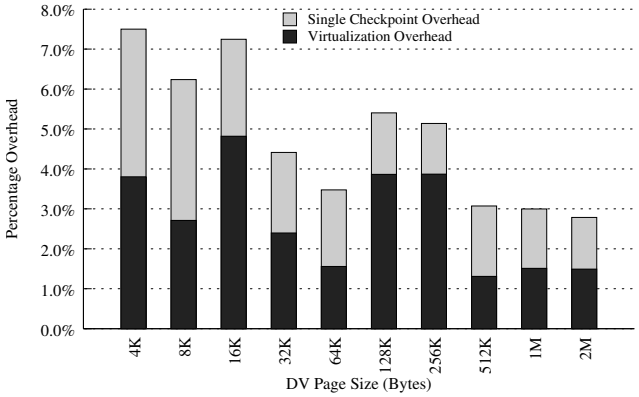

Figure 3. Comparison of DejaVu overhead for Chombo vs. normal execution without DejaVu

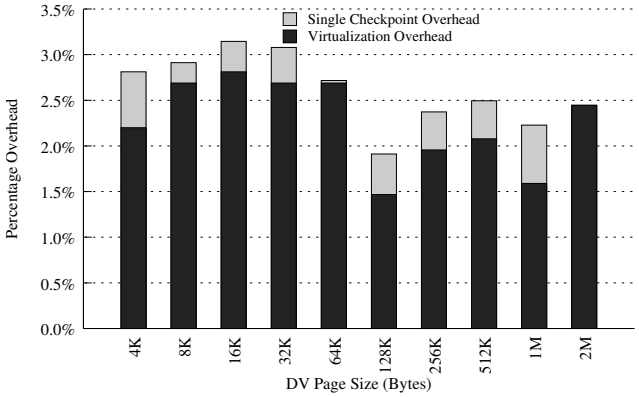

Figure 4. Comparison of DejaVu overhead for Sander vs. normal execution without DejaVu

than 5%, which is relatively low for a transparent checkpoint recovery system [13].

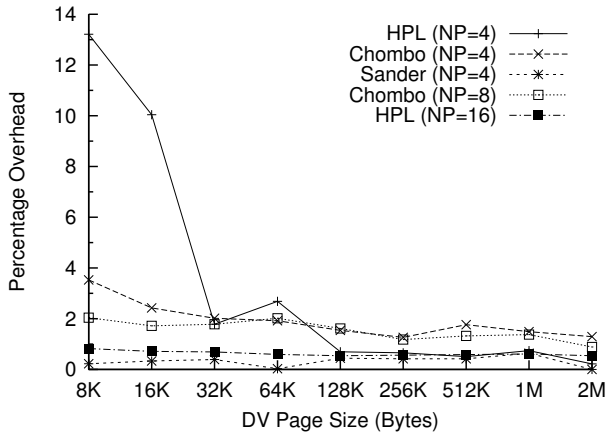

Figure 5. Checkpoint overhead of varying DejaVu page size for several applications

Figure 5 shows the checkpoint overhead on the performance of High Performance Linpack, Chombo, and Sander running with between 4 and 16 processors. The problem size of the NP=4 HPL run was set to a 22,000 x 22,000 matrix so the total memory size of the job was at least 3.87 gigabytes; the NP=16 HPL run was set to a 32000 x 32000 matrix with total size of 8.19 gigabytes. The problem size for the Chombo runs was a 64 element cell that resulted in 9.6 gigabytes of total memory. The Sander run had a smaller memory footprint as that application is mostly CPU bound. The HPL runs were configured to run for approximately 50 minutes so the results reflect the expected checkpoint overhead with a checkpoint interval of once an hour. The Chombo and Sander tests ran for about 15 minutes and the results were extrapolated to show the same overhead amortized over a 60 minute period, or one checkpoint per hour. Figure 5 shows the percentage overhead with respect to the runtime configurable page size used in DejaVu. The overhead for a 4KB page size was proportionally much higher and has been omitted from figure 5 so that the remaining data points are visible. It is clear from figure 5 that page sizes greater than 64KB provide the best performance in terms of reduced overhead. The checkpoint overhead at page sizes greater than 64KB was less than 2.0%. The larger page sizes reduce the number of page faults that the application must incur. For an application like HPL that has high spatial locality this larger page size results in greater performance. Additionally, on fast parallel file systems like GPFS the optimal block size for file operations is typically much larger than the default 4k operating system page size. Increasing the DejaVu page size allows DejaVu to perform larger writes to the checkpoint file which improves file I/O performance.

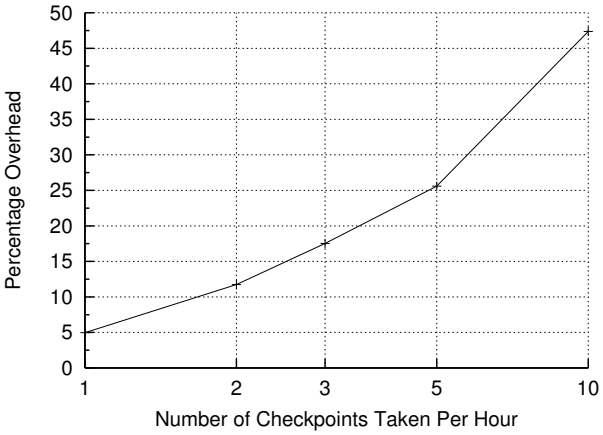

Figure 6. Overhead of increased checkpoints per hour for Chombo

To illustrate the cost of decreasing the checkpoint interval, we have measured the cost of performing an increasing number of checkpoints within a single time period as shown in figure 6. The results show that increasing the number of checkpoints taken per hour increases the checkpoint overhead; in fact, the increase is nearly linear for Chombo. Of course, these results depend heavily upon how frequently the application will dirty memory between the incremental checkpoints. Chombo performed frequent large memory allocations and deallocations that caused a large amount of the memory space to become dirty between checkpoints. Therefore, each incremental checkpoint required a significant number of memory regions to be saved to disk thereby increasing the overhead of the frequent checkpoints. If an application modified memory at a slower pace, then overhead of frequent interval checkpointing would improve. Therefore, a periodic checkpointing system must choose a reasonable checkpoint interval that provides the maximum efficiency from restart capability while not degrading overall system performance from overly-frequent checkpointing.

The overhead from a restart operation has not been included in this evaluation since it is a relatively infrequent operation compared to the common case of a correctly executing system[15]. The overhead of application virtualization and checkpointing represent the major portion of the runtime costs for a checkpoint/restart system.

6 Conclusions and Future Work

Studies [8] have shown that as distributed systems continue building up towards the peta-scale class, rollback-

recovery mechanisms will become an increasingly necessary component of efficient deployments. In this paper, we presented DejaVu, a new fault tolerance system for transparent and automatic checkpointing, migration and recovery of parallel and distributed applications. It is a concurrent, incremental checkpointing system and includes a recoverable IO subsystem. It is transparent to applications, the operating system, and does not require specialized MPI implementations for socket based networks. We implemented the DejaVu system for AMD64 based distributed systems running the GNU/Linux operating system and interconnected over Infiniband. Results from a performance evaluation over a range of real world applications/benchmarks shows that the overhead of DejaVu is suitable for deployment in production systems. We found in all cases that with some tuning of DejaVu's runtime behavior, the overhead of checkpointing a distributed system with per-process memory footprints in the GB+ range is below 5%.

We are currently developing an implementation of DejaVu using MPICH-MX for native performance on Myrinet networks and we are also exploring new unified methods of virtualizing MPI libraries transparently, regardless of the underlying interconnect technology. Finally, we are developing a queuing system called DejaQu that integrates seamlessly with DejaVu enabling preemptive scheduling of computing resources.

References

[1] G. Bosilca and et al. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Washington, DC, USA, 2002. IEEE Computer Society.

[2] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level checkpointing for shared memory programs. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 235–247, New York, NY, USA, 2004. ACM Press.

[3] D. Case, T. Darden, T. Cheatham III, C. Simmerling, J. Wang, R. Duke, R. Luo, K. Merz, B. Wang, D. Pearlman, M. Crowley, S. Brozell, V. Tsui, H. Gohlke, J. Mongan, V. Hornak, G. Cui, P. Beroza, C. Schafmeister, J. Caldwell, W. Ross, and P. Kollman. AMBER 8. *University of California, San Francisco*, 2004.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[5] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, and B. V. Straalen. Chombo software package for AMR applications design document. Technical report, Applied Numerical Algorithms Group, NERSC Division, Lawrence Berkely National Laboratory, September 2003.

[6] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[7] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart, 2002.

[8] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, April-June 2004.

[9] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[10] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[11] H. Jung, D. Shin, H. Han, J. W. Kim, H. Y. Yeom, and J. Lee. Design and implementation of multiple fault-tolerant MPI over Myrinet (M^3). In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 32, Washington, DC, USA, 2005. IEEE Computer Society.

[12] K. Koch. How does asci actually complete multi-month 1000-processor milestone simulations?, April 2002.

[13] K. Li, J. Naughton, and J. Plank. Low-latency, concurrent checkpointing for parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 5(8):874–879, August 1994.

[14] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[15] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scal cluster systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005.

[16] D. Patterson and et al. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, U.C. Berkeley, Computer Science Department, 2002.

[17] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. Technical report, University of Tennessee, 1994.

[18] D. Reed. High-end computing: The challenge of scale. director's colloquim, May 2004.

[19] 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 *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.