Kai Hwang and Hai Jin
University of Southern California

Edward Chow and Cho-Li Wang
University of Hong Kong

Zhiwei Xu
Chinese Academy of Sciences

Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space

Adopting a new hierarchical checkpointing architecture, the authors develop a single I/O address space for building highly available clusters of computers. They propose a systematic approach to achieving single system image by integrating existing middleware support with the newly developed features.

The computing trend is moving from clustering high-end mainframes to clustering desktop computers. This trend is triggered by the widespread use of PCs, workstations, gigabit networks, and middleware support for clustering.[1] This article presents new approaches to achieving fault tolerance and *single system image* (SSI) in a workstation cluster. In a cluster of computers, local area networks or high-bandwidth switch networks using optical fibers physically connect a collection of node computers. The workstations in a cluster can work collectively as an integrated computing resource—that is, an SSI—or they can operate as individual computers, separately.

Present clusters are usually small and provide only limited SSI services. Future clusters will likely increase in scalability and offer more SSI support, as Figure 1 illustrates. The implication is that future clusters could replace the MPP, SMP, or CC-NUMA architectures (see "The cluster as a computer architecture" sidebar for key characteristics of these computer platforms).

We focus on clusters with high availability through SSI support, distributed RAID (redundant arrays of inexpensive disks) with parity checks, and hierarchical checkpointing with adaptive recovery. In particular, we developed a single I/O address space among all disks and peripheral devices attached in the cluster. This enables direct remote disk access, which is a necessary step to implement a

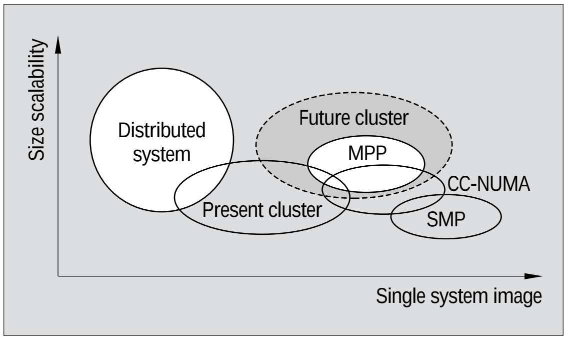

Figure 1. Design space of competing computer architectures.

1092-3063/99/$10.00 © 1999 IEEE

The cluster as a computer architecture

Different research groups have referred to the term “clusters” as multicomputer clusters,[1] clusters of workstations (COW),[2] or networks of workstations (NOW).[3] To compare clusters with other competing computer systems, Table A summarizes four major competing classes of computer architectures.

These classes include massively parallel processors, symmetric multiprocessors with shared memory, and scalable multiprocessors having a cache-coherent nonuniform memory access architecture (CC-NUMA). Distributed systems are the conventional network of independent computers. Because each node runs with its own operating system, a traditional network of computers has multiple-system images on different nodes.

An SMP server must have a single system image with a centralized shared memory. In a cluster, having SSI across all computer nodes is desirable.[4] The distributed systems and SMP are two extreme architectures with respect to SSI. The cluster, MPP, and CC-NUMA are computer architectures between the two extremes.[5] Kai Hwang has accessed various network technologies for building clusters.[6]

Internet-based metacomputing[7] uses a large number of remote hosts from the Internet to form a supercluster. Any PC or workstation user can utilize the supercluster’s resources. However, metacomputing cannot be easily arranged because of the ownership problem associated with scattered workstations and PCs. Metacomputing demands even more software support to manage and coordinate the huge heterogeneous computing resources on the Internet for collective applications.

References

1. K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, WCB/McGraw-Hill, New York, 1998.
2. G.F. Pfister, *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing,* 2nd ed., Prentice Hall PTR, Upper Saddle River, N.J., 1998.
3. T.E. Anderson et al., “A Case for NOW (Networks of Workstations),” *IEEE Micro*, Vol. 15, No. 1, Feb. 1995, pp. 54–64.
4. G.F. Pfister, “The Varieties of Single-System Image,” *Proc. IEEE Workshop on Advances in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 59–63.
5. M.A. Baker, G.C. Fox, and H.W. Yau, *Cluster Computing Rev.* NPAC Tech. Report SCCS-748, Northeast Parallel Architectures Center, Syracuse Univ., Syracuse, N.Y., 1995.
6. K. Hwang, “Gigabit Networks for Scalable Multiprocessors and Multicomputer Clusters,” *Trans. Hong Kong Inst. of Eng.*, Vol. 2, No. 3, Dec. 1997, pp. 82–87.
7. C. Catlett and L. Smarr, “Metacomputing,” *Comm. ACM*, Vol. 35, No. 6, June 1992, pp. 44–52.

Table A. Key characteristics of scalable parallel computers.[1]

Characteristic	MPP	SMP, CC-NUMA	Cluster	Distributed system
Number of nodes	100–1000	10–100	100 or less	10–10,000
Node complexity	Fine or medium grain	Medium or coarse grain	Medium grain	Wide range
Internode communication	Message passing or shared variables for DSM	Shared memory	Message passing	Shared files, RPC, message passing IPC protocol
Job scheduling	Single-run queue at host	Single-run queue mostly	Multiple queues	Independent multiple queues
SSI support	Partially	Always in SMP and some NUMA	Desired	No
Node OS copies and type	*N* microkernels and 1 monolithic OS at host	One monolithic OS for SMP, multiple OSs for NUMA	*N* OS platforms (homogeneous or microkernel)	*N* OS platforms (heterogeneous)
Address space	Multiple (single for DSM)	Single	Multiple or single	Multiple
Internode security	Unnecessary	Unnecessary	Required if exposed	Required
Ownership	One organization	One organization	One or more organizations	Many organizations
Network protocol	Nonstandard	Nonstandard	Standard	Standard
System availability	Low to medium	Low for SMP higher for NUMA	Highly available or fault tolerant	Medium
Performance metric	Throughput and turnaround time	Turnaround time	Throughput and turnaround time	Response time

reliable cluster of workstations or build robust PC clusters.

SSI and availability

A cluster should have SSI and availability supported by middleware between the node OS and user application environment. The middleware consists of essentially two layers of software, which glue all node OSs together to establish SSI and enhanced availability. The availability infrastructure enables the cluster services of checkpointing, automatic failover, recovery from failure, and fault-tolerant operation among all cluster nodes.

The SSI layer supports collective cluster applications, which demand operational transparency and scalable performance. The clusters offer SSI at a wide range of abstraction levels. At one extreme, a cluster can function as a tightly coupled SMP, NUMA, or MPP system; at the other extreme, it can behave like distributed computer systems with multiple system images.

Design goals

Cluster design goals commonly focus on *complete transparency* in resource management, *scalable performance*, and *enhanced availability* in supporting user applications.

Complete transparency

The SSI layer should let the user see a single cluster system instead of a collection of independent nodes. For example, in an SSI cluster with a single entry point, users can log in from any node. The needed software is stored on one node only. In a loosely coupled network of computers, users must install the same software for each node. To achieve complete transparency in resource allocation, deallocation, and replication, the implementation details should be invisible to user processes.

To have a single entry point, when a user telnets and ftps to the cluster, the two sessions should see the same home directory. The user wants to use a single file hierarchy, but the way the file system is physically organized in the backup storage is transparent to the user. The file system can be mounted from a central file server or fully distributed among many nodes, and a file can be duplicated in several nodes. When a node fails, the portion of the file system can migrate to another node.

Scalable performance

The scalability is related to the cluster's performance. An efficient cluster should not be limited in physical size and loading pattern. When a node is added to a cluster, the protocol and API (application programming interface) need not change. The cluster's performance should scale with more nodes allocated, and the SSI services must include load balancing and parallel support. For example, a single entry point should distribute a login request to the least-loaded node. Cluster efficiency also demands that all SSI services have small overheads. The time to execute the same operation on a cluster should not be much longer than that on a single workstation.

Enhanced availability

In a cluster, the SSI services should be highly available at all times. Any single point of failure should be recoverable without affecting a user's applications. So, high availability should employ checkpointing and fault-tolerant technologies to enable rollback *recovery*. A fault-tolerant cluster automatically demands the features of *hot standby*, *failover*, and *failback* services after a node failure. Hot standby refers to the situation that a primary node provides the service, while a back-up node stands by without doing any work. The standby node is ready to take over the work as soon as the primary node fails. Failover means that the surviving nodes take over the services originally provided by the failed node. Failback allows the failed node to rejoin the workforce after it is repaired.

Replicating resources creates a consistency problem—for example, with file-system coherency and shared-memory consistency. When the cluster performs multiple operations, it should be able to keep the duplicated data objects consistent. Critical section resources must be properly locked or protected to achieve data integrity. Other availability issues include security and data encryption to protect access to cluster resources.

Desired services and functions

We first identify all useful SSI services and availability functions. An example cluster configuration illustrating the key concepts follows.

SSI services

The following fundamental services stretch along different dimensions of the application domain, yet the services are mutually supportive.

- *Single entry point*: A user logs in to the cluster as a single system (for example, *telnet cluster.mycompany.com*), instead of logging in to individual nodes as in a LAN (local area network) environment (for example, *telnet node1.cluster.mycompany.com*).
- *Single file hierarchy*: Once logged in, the user sees a single hierarchy of file directories under the same root directory, just like a single file-management environment for workstation users. Examples of single file hierarchy include NFS (network file system), AFS (Andrew file system), xFS (serverless file system), and Solaris MC (multicomputer) Proxy.[2]
- *Single control point*: The entire cluster is managed from a single place using a single GUI tool, much like the IBM AIX workstation managed by a special software package called SMIT.
- *Single virtual networking*: Any node can access any network connection throughout the cluster domain. Multiple networks support a single virtual network operation.
- *Single memory space*: This service gives users the illusion of distributed shared memory over local memories physically distributed over the cluster nodes. Distributed shared memory enables shared-variable programming. TreadMarks[3] gives the best example of software-implemented DSM.
- *Single job-management system*: Under a global job scheduler, a user job can be submitted from any node to request any number of host nodes to

Table 1. Four middleware packages for SSI and availability services.

SUPPORT FEATURES	GLUNIX	TREADMARKS	CODINE	LSF
Single control point	Yes	No	Yes	Yes
Single entry point	Yes	No	No	No
Single file hierarchy	Yes	Yes	Yes	Yes
Single memory space	No	Yes	No	No
Single process space	Yes	No	No	No
Single I/O space	No	No	No	No
Single networking	No	No	No	No
Batch support	Yes	No	Yes	Yes
Interactive support	Yes	Yes	Yes	Yes
Parallel support	Yes	Yes	Yes	Yes
Load balancing	Yes	No	Yes	Yes
Job monitoring	Yes	No	Yes	Yes
Suspend/resume	No	No	Yes	Yes
Dynamic resource	Yes	No	Yes	Yes
User interface	cmd-line	cmd-line	GUI/cmd-line	GUI
Checkpointing	No	No	Yes	Yes
Process migration	No	No	Yes	Yes
Security standard	Unix	Unix	Kerboros	Kerboros
Fault tolerance	Yes	No	Yes	Yes

execute it. Concurrent job scheduling is possible in batch, interactive, or parallel modes. Examples of job-management systems for clusters include GLUnix (Global Layer Unix),[4] LSF (load-sharing facility),[5] and Codine (*http://www.genias.de/genias/english/codine/Welcome.html*).

- *Single user interface*: The users should be able to use the cluster through a single graphic interface. Such an interface is available for workstations (for example, the Common Desktop Environment) and PCs (for example, Microsoft Windows 95). To develop a cluster GUI, you can use Web technology.

AVAILABILITY SUPPORT FUNCTIONS

Listed below are three desired features to enhance the availability of clusters of workstations or PC clusters:

- *Single I/O space*: This function allows any node to remotely access any I/O peripheral or disk devices without the knowledge of their physical location. In our SIOS design, all distributed disks, RAIDs, and devices form a single address space.
- *Single process space*: This requires mutual understanding between processes created in the same address space. They share a uniform process-identification scheme. A process on any node can create (for example, through a Unix fork) or communicate with (for example, through signals or pipes) any other process on a remote node.
- *Checkpointing and process migration:* Checkpointing is a software mechanism to periodically save the process state and intermediate computing results in memory or on disks. This allows rollback recovery after a failure. Process migration is needed in dynamic load balancing among the cluster nodes and in supporting checkpointing.

A CLUSTER EXAMPLE

Our example cluster has four host nodes, two of which are connected with I/O devices attached. A properly designed cluster should behave like one single system. In other words, it behaves like a giant workstation with four network connections and four I/O devices attached. Any process on any node can use any network and I/O devices, such as RAIDs or CD-ROM drives.

This cluster is a Web server. The Web information database is distributed between two RAIDs. An HTTP daemon is started on each of the nodes to handle the Web requests, which come from four different network connections to the four nodes. The SIOS implies that any node can access the RAIDs. Suppose most requests come from an ATM network. It would be beneficial if we could distribute the http functions to all four nodes.

For single point of control, the system administrator can configure, monitor, test, and control the entire cluster and each individual node from a single point. Many clusters achieve this through a system console that connects to all cluster nodes. The system console normally connects to an external LAN so that the administrator can remotely log in to the system console. Single point of control does not mean that the system console solely carries out all system administration work. Instead, the entire cluster is managed from a single place using a single GUI tool.

Middleware support for SSI and availability

Cluster design concerns size scalability, enhanced availability, system manageability, fast message passing, security and encryption, and distributed computing environments. Table 1 summarizes four representative middleware packages for SSI and availability support. GLUnix is available in the public domain, and the other middleware packages are commercial products. The first seven features support SSI services, the next eight features facilitate job management and parallel programming, and the remaining four features are for availability and fault tolerance.

Batch support refers to the dedicated use of cluster resources by a single user job in a batch mode. *Interactive support* refers to the time-sharing use of cluster resources by multiple users simultaneously. *Parallel support* refers to the management of parallel processes and MPI and PVM environments. So far, none of the four middleware packages have implemented SIOS and single networking features.

Process migration is needed to achieve dynamic load balancing among cluster nodes. The process running on a failing host can be switched (*failover*) to a surviving host in the cluster. Process migration enables load distribution, fault resilience, system administration, and improved data-access locality. Migrating processes from overloaded nodes to lightly loaded ones can achieve load distribution. Migrating processes from nodes that might have partial failure can achieve fault resilience.

Job monitoring, suspend/resume, dynamic resources, and user interfaces are useful features that provide a user-friendly environment in program debugging,

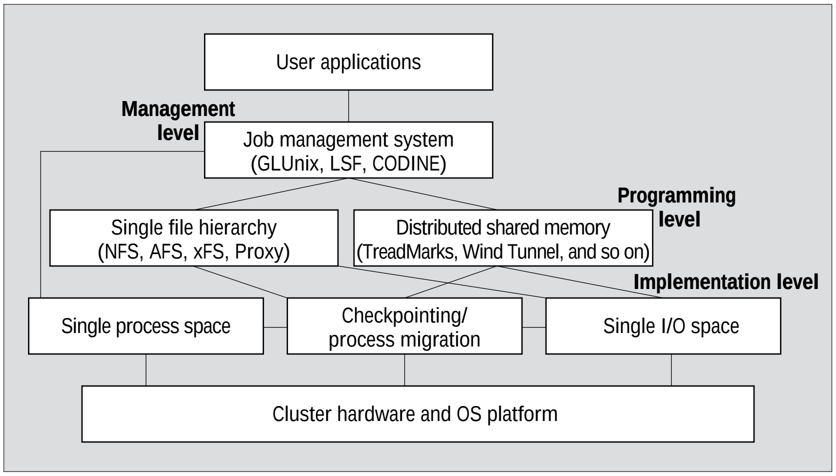
User applications
Management level
Job management system (GLUnix, LSF, CODINE)
Programming level
Single file hierarchy (NFS, AFS, xFS, Proxy)
Distributed shared memory (TreadMarks, Wind Tunnel, and so on)
Implementation level
Single process space
Checkpointing/ process migration
Single I/O space
Cluster hardware and OS platform

Figure 2. Relationships among middleware modules for clusters.

performance evaluation, resource allocation, and program optimization. Fault tolerance and checkpointing are features to enhance cluster-system availability. To achieve transparency, the job-management system should be able to reconfigure the cluster dynamically with minimal impact on the running jobs.

The functionality of a job-management system is often distributed. For instance, a user server can reside in each host node, and the resource manager can span over all the cluster nodes. The system can migrate processes from the nodes that are about to be shut down so that the system administrator doesn't need to wait until a user logs out. Migrating processes toward the source of the data can also improve the data-access locality.

GLUnix at Berkeley

GLUnix can be easily downloaded to any cluster through the public domain. This global layer provides an SSI of the nodes in a cluster so that all of the processor, memory, network capacity, and disk bandwidth can be allocated for sequential and parallel applications. The global layer is realized as a protected, user-level operating-system library that is dynamically linked to every application. The library intercepts all system calls and recognizes them as procedure calls.

The GLUnix package has three distinct advantages:

- Implementing and modifing its source code at the user level is easy.
- It supports coscheduling of parallel programs, idle-resource detection, process migration, load balancing, remote paging, and some availability support.
- It was designed to port to any OS that supports interprocess communication, process signaling, and access to loading information.

TreadMarks at Rice

To enable shared-memory computing on Norma (*no remote memory access*) and non-CC-NUMA systems, researchers have proposed the software-coherent NUMA memory model, also known as the DSM model. The TreadMarks project at Rice University[3] has developed a runtime library of software routines to implement DSM with a single address space, data sharing, and memory consistency over a cluster of Unix workstations.

Codine and DQS

The Codine package evolves from the *Distributed Queuing System* (DQS) created at Florida State University. Codine is offered by Genias Software GmbH in Germany. Genias claims that this package has become a de facto job-management system in Europe. The major strength of Codine lies in hardware and software resource management in a heterogeneous networked environment. Codine uses GUI tools to provide a SSI of cluster resources. Checkpointing is supported only if the kernel supports it. Kernel-checkpointed jobs can migrate, and resources can be dynamically added or deleted from a resource pool.

LSF

The LSF package from Platform Computing evolves from the Utopia system developed at the University of Toronto. LSF emphasizes job management and load sharing of both parallel and sequential jobs. In addition, it has support for checkpointing, availability, and load migration. LSF has been implemented on various UNIX and Windows/NT platforms.[5] Checking the entries in Table 1, Codine and the LSF are almost equally capable in supporting the same SSI and availability functions.

Middleware packages for clusters

In Figure 2, we show the functional relationships among six key middleware packages. These middleware packages are used as interfaces between user applications, cluster hardware, and the OS platform. They support each other at the management, programming, and implementation levels.

The job-management system is essentially a global job scheduler. The single file hierarchy and DSM support distributed file management and shared-memory programming. The single process system, checkpointing, process migration, and SOIS help implement the job-management system, single file hierarchy, and DSM services.

All middleware packages work together to support the desired availability and SSI services. The SIOS module enables the efficient implementation of DSM, single file hierarchy, and checkpointing/process-migration functions.

Distributed RAID and cluster architectures

Here we assess three architectural design options for enhancing the availability and fault tolerance of a cluster of workstations or PCs.

RADD

M. Stonebraker and G. Schloss first proposed the *Redundant Arrays of Distributed Disks* architecture as a multicopy algorithm for distributed RAID systems.[6] All local disks, attached to different cluster hosts, logically form the RADD subsystem. Normally, it stores the checkpointing data in local disk blocks sequentially, while parity blocks reside in other local disks.

Among different nodes, the RADD applies the RAID-5 algorithm to handle

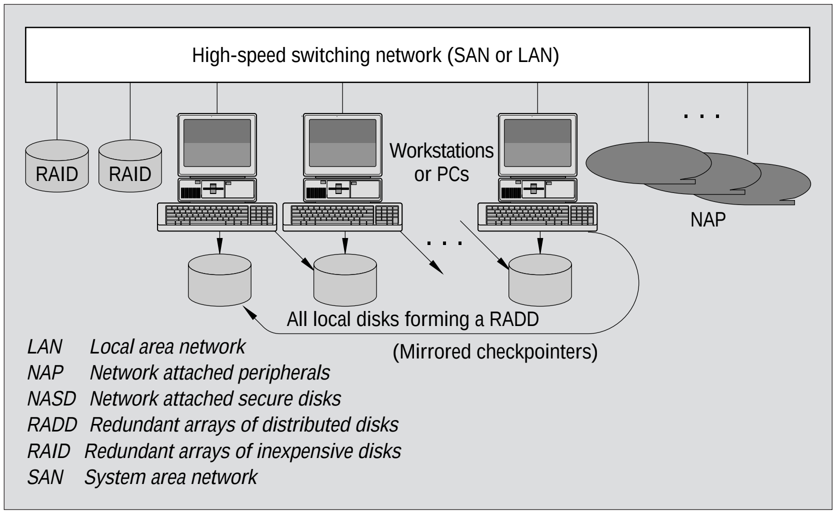
High-speed switching network (SAN or LAN)
. . .
RAID
RAID
Workstations or PCs
NAP
. . .
All local disks forming a RADD
(Mirrored checkpointers)
LAN Local area network
NAP Network attached peripherals
NASD Network attached secure disks
RADD Redundant arrays of distributed disks
RAID Redundant arrays of inexpensive disks
SAN System area network

Figure 3. A fault-tolerant cluster architecture with distributed local disks (RADD), network-attached secure disks (NASD), and network-attached peripherals (NAP) forming a single I/O space.

local I/O operations, which are transparent to higher-level RADD operations. For simplicity, you can readily apply the RAID-1 architecture on local disks. RADD implements mirroring on neighboring disks, but there is no parity among the distributed local disks.

NASD

Network-Attached Secure Disks[7] attach the RAIDs directly to the network as a stable storage to allow shared access by all cluster nodes. Each workstation node in the cluster might or might not have a local disk attached to it. Even with locally attached disks, the nodes buffer the data retrieved from the NASD. The NASD supports independent accesses by all nodes. Thus, a specially designed NASD controller must resolve the access conflicts.

The NASD architecture is quite different from the server-attached RAID. In the latter case, block data transfer must be done through the network server instead of directly from the network to end users at local workstations. NASD improves its scalability by removing the bottleneck problem from the network server. A technique called *network striping* makes this possible.[8]

Checkpointing cluster architecture

Figure 3 illustrates the checkpointing cluster architecture we originally proposed at the University of Hong Kong and have continued to study at the University of Southern California. The cluster nodes are either workstations or PCs. A Gigabit LAN or a SAN (system area network)[1] connects all the nodes, and local disks are attached to each workstation node. Each local disk is accessible only from its own attached host, and all the local disks form a RADD.

The network-attached RAIDs form a NASD as the stable storage for implementing various checkpointing schemes. We use independent checkpointers over the coordinated ones to reduce the checkpoint overhead and recovery latency. This cluster design's uniqueness lies in its distributed-checkpointing RAID architecture. It is based on the three-level adaptive recovery scheme and the SIOS we propose in this article.

Hierarchical checkpointing schemes

We apply three checkpointers for designing checkpointing schemes. In these schemes, mirroring is applied to reduce the probability of rolling back to the stable storage. The three types of checkpointers are

- 1*-checkpointer (local-disk checkpointer)*: The processes periodically store checkpoints in their own local storages. This type of checkpointer can tolerate only a transient processor failure. When a permanent processor failure or a local disk failure occurs, the local disk is inaccessible and the 1-checkpointer is lost. The 1-checkpointer can be implemented in either coordinated or independent checkpointing schemes.
- M*-checkpointer (mirrored checkpointer)*: The processes periodically save consistent checkpoints to their local disks and copy the mirrored images to their neighbor's disk. This type of checkpointer can tolerate single failures and multiple, isolated, permanent failures.
- N*-checkpointer (stable-storage checkpointer)*: The processes periodically save consistent checkpoints on the stable storage. Because stable storage is assumed to be failure-free, this type of checkpointer can tolerate any number of failures.

Adaptive recovery levels

Our checkpointing schemes offer adaptive two- or three-level rollback recovery with a reduced recovery latency. Figure 4 illustrates the recovery at three levels. The three levels of rollback recovery are

- *Level 1*: Rollback to a 1-checkpoint, when a processor has a transient failure that does not immediately follow an *M*-checkpoint or an *N*-checkpoint.
- *Level 2*: Rollback to an *M*-checkpoint, when a node has a permanent failure, assuming no adjacent failures occur and they do not follow an *N*-checkpoint.
- *Level 3*: Rollback to the stable storage checkpoint (*N*-checkpoint), when a failure occurs immediately following an *N*-checkpoint or when a processor or a local disk has a permanent failure and loses its mirrored checkpoint.

In the worst case, the loss of useful computation for a level-two recovery is *mT*, while that for a level-three recovery is *mnT*, where *T* is the checkpointing period. With a much higher recovery latency contingent on network latency and simultaneous access of the central stable storage, the level-three recovery latency should be much larger than the level-two recovery. This reduces the probability of an *N*-rollback to the stable storage. The level-two recovery can shorten the expected recovery latency significantly, if properly implemented.

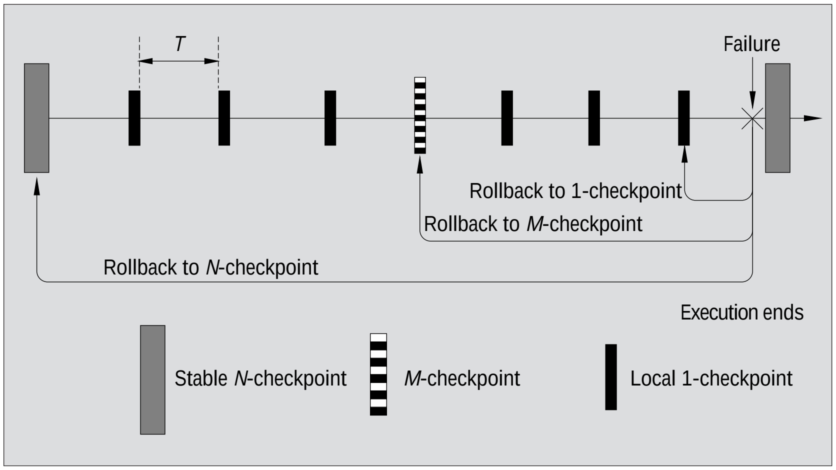

Figure 4. Adaptive rollback recovery at three levels in the distributed cluster disk hierarchy.

Checkpointing schemes

Checkpointing schemes can be implemented with one to three levels of recovery. The simplest single-level checkpointing scheme involves remote memory or local disks. J.S. Plank and his colleagues developed diskless checkpointing in remote memory.[7] Alternatively, you can save the 1-checkpointer periodically on a local disk. The major drawback of this one-level scheme is its limited fault coverage. A single permanent failure will paralyze the node by losing the checkpoint file, preventing any chance of a rollback recovery.

N.H. Vaidya proposed a two-level recovery scheme that can tolerate a local disk crash because of the allowed rollback to an *N*-checkpointer in a stable storage.[9] The major drawback is its large *N*-checkpoint overhead and recovery latency. While the recovery latency of this two-level rollback scheme improves sharply over the one-level scheme, we introduce the *M*-checkpointer to implement improved two-level or three-level schemes.

The *M*-checkpointer tolerates at least single processor failure, and it can potentially tolerate conditional multiple permanent failures, corresponding to the scenario in which failures do not occur between mirrored workstations. If the above condition is satisfied, no checkpoint file is lost and the spare workstation can be used to have a full recovery.

For a cluster of *N* workstations, the *M*-checkpointer can tolerate at most *N*/2 permanent failures. The *M*-checkpointer has a latency between the other two checkpointers. The idea is to save every *m*th checkpoint as an *M*-checkpoint and/or every *mn*th checkpoint as an *N*-checkpoint. Figure 5 shows the timing charts of two hierarchical checkpointing schemes. The checkpointer overhead corresponds to the bar width in the charts.

Scheme A involves two-level interleaved mirror and stable checkpointing. It saves the *M*-checkpoints in mirrored disks and *N*-checkpoints in the stable storage (see Figure 5a). Every *m*th consistent checkpoint is stored in the stable storage, while all other checkpoints are stored in local memories (instead of local disks) with a mirrored image on a neighbor's disk. This scheme rolls back to a local memory for transient failures.

For permanent failures, the scheme rolls back to a mirrored copy of the checkpoint. It rolls back to the stable storage only when a permanent failure immediately follows an *N*-checkpoint. This scheme provides the lowest recovery latency.

Scheme B involves three-level adaptive checkpointing and recovery. Figure 5b illustrates this scheme, in which the host processor periodically saves the 1-checkpoints in local disk. It saves every *m*th checkpoint as an *M*-checkpoint. It stores every *mn*th consistent checkpoint

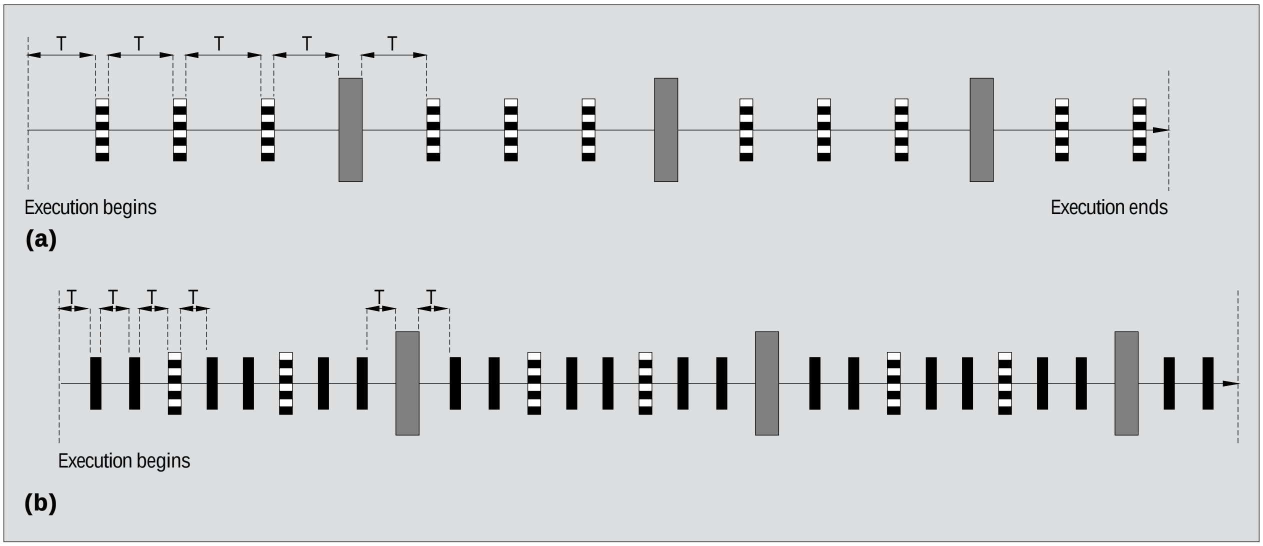

Figure 5. Two hierarchical checkpointing and recovery schemes (*T* = checkpointing period): (a) Scheme A: two-level mirror and stable checkpointing recovery; (b) Scheme B: three-level adaptive checkpointing.

on the stable storage, while saving all other checkpoints on local disks.

Overhead versus latency

Denote P_1, P_M, and P_N as the probabilities of rolling back to a 1-checkpoint, an M-checkpoint, and an N-checkpoint. Let C_1, C_M, and C_N be the corresponding checkpointing overheads. Let R_1, R_M, and R_N be the repeated computation times. Finally, let L_1, L_M, and L_N be the total recovery latencies, respectively. Let T be the fixed time interval when coordinated checkpointing is employed. We know that, within a given time interval, the following inequalities hold: $C_1 < C_M < C_N$, $P_1 > P_M > P_N$, and $L_1 < L_M < L_N$.

The situation is similar to a two- or three-level memory hierarchy (cache, L2 cache, and main memory), with P analogous to the hit ratio and R analogous to the memory-access latency. On the average, a multiple-level checkpointing and recovery scheme can achieve shorter latency over the single recovery scheme, because it makes the common case fast.

The insertion of the M-recovery reduces the probability P_N of invoking an N-recovery significantly. For example, let $p = 0.01$ be the probability of a permanent failure within a given time interval and $n = 8$ between two adjacent N-checkpoints. A permanent failure requires a rollback to an N-checkpoint with a probability $P_N = 1 - (1 - p)^n = 1 - 0.99^8 = 0.077$. With the M-recovery, only the scenario with permanent failures that are adjacent require a rollback to an N-checkpoint. Thus, the probability $P_N = 1 - (1 - p^2)^n = 0.00079$, about two orders of magnitude smaller. In general, adding the level of M-checkpoint recovery reduces P_N by at least two to three orders of magnitude.

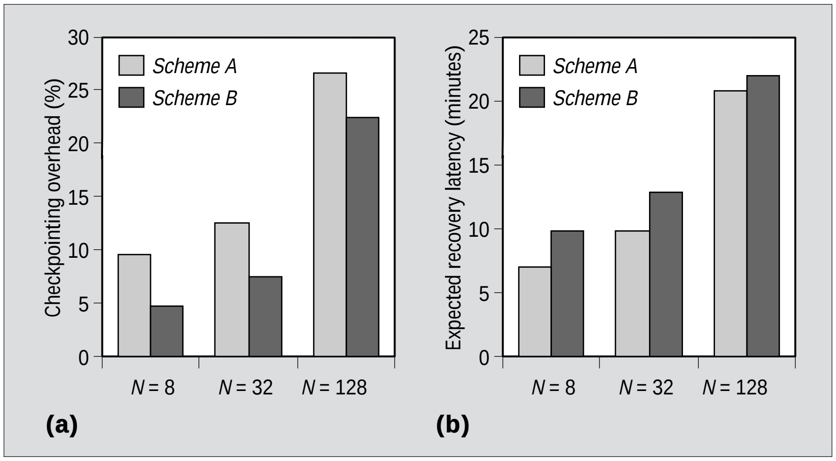

Figure 6. (a) Checkpointing overhead and (b) recovery latency between two schemes.

We calculate the total *expected recovery latency* as $P_1 L_1 + P_M L_M + P_N L_N$, where $P_1 + P_M, + P_N = 1$. In Table 2, we show the two formulae for Schemes A and B.

Consider a program of a total execution time $G = 20$ hours. Assume the following parameter values based on past experience: $T = 5$ minutes, $C_1 = 10s$, $R_1 = 15s$, $C_M = 25s$, $R_M = 15s$, $C_N = 15Ns$, $R_N = 15Ns$, $m = 8$, $n = 4$, and $p = 0.02$, where s stands for second. The *checkpointing overhead percentage* is the percentage of the total checkpointing overhead over the total execution time without failure. Figure 6a plots the COP of the two checkpointing schemes for clusters, with N ranging from 8 to 128 nodes.

Scheme A saves mostly the mirrored checkpoints and has a higher COP than that of scheme B. In both schemes, the COP increases steadily with respect to the cluster size. Figure 6b compares the expected recovery latency of the two recovery schemes in the worst case. Scheme A has a shorter recovery latency, because rolling back to an M-checkpoint at the neighboring disk is much faster than rolling back to an N-checkpoint. All latencies are independent of the total execution time G.

Scheme B takes slightly longer than Scheme A to recover from a failure. This is because Scheme B takes longer to recover from an M-checkpoint, as revealed in the L_M term. For large cluster sizes, Schemes A and B perform almost equally, because P_M increases faster in Scheme B than in Scheme A, as the machine size increases to 128 nodes.

All previous studies have shown that writing the state of a process to stable storage contributes the most to the total latency. The simplest way to save the process state is to suspend it, save its address space on storage devices, and then resume it. This scheme can be costly with M-checkpointing and N-checkpointing programs with a large address space.

Concurrent checkpointing[10] does not suspend the execution of the process while the checkpoint is saved on the storage devices. It relies on the memory-protection hardware that is commonly available in a modern computer. Adding *incremental checkpointing* can further reduce the overhead. Incremental checkpointing avoids rewriting portions of the process states that do not change between consecutive checkpointings.

Single I/O space design

Sun Microsystems has extended the Solaris OS for clustering Sun workstations. This extension is called Solaris MC. They use a uniform device-naming scheme to achieve SIOS in addressing any peripheral or disk devices attached to a Unix cluster of Sun

Table 2. The upperbound on the expected recovery latency for Schemes A and B.

Schemes	Upper bound
Scheme A	$P_1(R_1 + T + C_1) + P_M(R_M + T + C_M) + P_N(R_N + m(n-1)C_1 + (m-1)C_M + C_N + mnT))$
Scheme B	$P_1(R_1 + T + C_1) + P_M(R_M + (m-1)C_1 + C_M + mT) + P_N(R_N + m(n-1)C_1 + (m-1)C_M + C_N + mnT)$

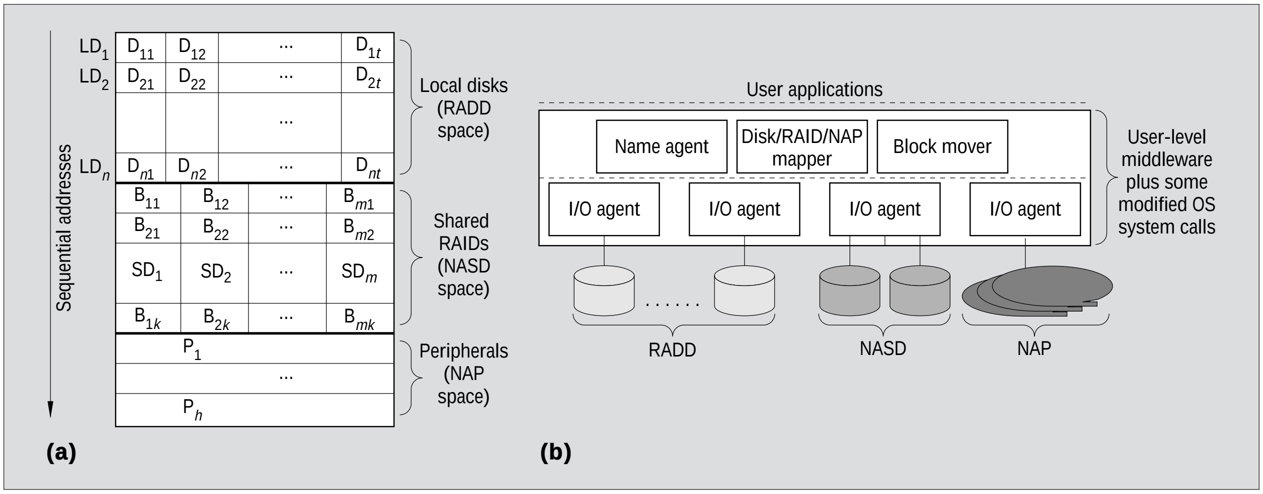
Sequential addresses
LD_1 D_{11} D_{12} ... D_{1t}
LD_2 D_{21} D_{22} ... D_{2t}
...
LD_n D_{n1} D_{n2} ... D_{nt}
Local disks (RADD space)
B_{11} B_{12} ... B_{m1}
B_{21} B_{22} ... B_{m2}
SD_1 SD_2 ... SD_m
B_{1k} B_{2k} ... B_{mk}
Shared RAIDs (NASD space)
P_1
...
P_h
Peripherals (NAP space)
(a)
User applications
Name agent
Disk/RAID/NAP mapper
Block mover
I/O agent
I/O agent
I/O agent
I/O agent
User-level middleware plus some modified OS system calls
......
RADD
NASD
NAP
(b)

Figure 7. Single I/O address space design: (a) the integrated I/O space and (b) addressing and mapping mechanisms.

workstations. A device address consists of a node number and the device number. A process can access any device by initiating a system call at a remote processor using this unique address. We consider the Solaris MC definition of SIOS a rather restricted one because it only accesses remote devices at the file level, not at the block or strip levels.

We developed a new SIOS concept.[11] SIOS is designed over all local disks, RAIDs, and I/O peripheral devices. We must consider hardware, software, and middleware requirements in the SIOS design, which must make remote access transparent without resorting to system calls. A local host, rather than the remote host, initiates the addressing process of remote devices. We implement SIOS primarily at the user level, but we must modify some local OS system calls to enable the remote disk accesses.

Advantages of the SIOS approach

Extending the SIOS design has four advantages. First, SIOS addressing eliminates the gap between accessing local disk and remote disks. Remote-disk access does not have to be handled as system calls from the remote host. Instead, the local host can address the remote disks directly by a modified system call.

Second, SIOS supports a persistent programming paradigm. DSM and SFH can be more easily implemented with the SIOS. All device types and their physical locations are now transparent to all users. MPI-based I/O currently cannot achieve this transparency.

Third, SIOS allows striping on remote disks, which accelerates parallel I/O operations often needed in moving large data files among the cluster nodes.

Finally, SIOS greatly facilitates the implementation of the distributed checkpointing and recovery scheme we suggested earlier. Mirroring on remote disks becomes faster and easy to control, and recovery from the shared RAIDs or NASD requires less time than current systems without SIOS.

The SIOS design at USC and HKU

As Figure 7a shows, the integrated I/O address space covers *n* local disks, *m* shared disks in the RAID, and *h* peripheral devices. The sequential address space is essentially a single, large, linear array of addresses down to the disk-block level or to the device-number level. For simplicity, we consider *t* blocks per local disk and *k* blocks per disk in the RAID. All the NAP devices are assigned high-order addresses in the linear array. These high-order addresses preserve the uniform naming scheme built in the Solaris MC.

Data mapping onto the distributed local disks uses parity blocks, mirroring, or shadowing. To improve the efficiency of I/O access, it stores user data sets on local disks and stores checkpointing or parity information on one or more remote disks. It also assumes sequential addresses horizontally in each local disk (LD_i) (see Figure 7a) and interleaves addresses in the shared RAID disks horizontally across the vertical disks (SD_j) in the RAID array. Hardware, firmware, or software can implement this address scheme.

Addressing and mapping mechanisms

Figure 7b shows the middleware functional modules needed to implement the SIOS for a cluster of workstations. The *name agent* maps the name known to the I/O agent onto the unique device number known to the Disk/RAID/NAP *mapper*. The mapper uses striping and replication to implement the address mapping.

The *block mover* copies data to and from the storage media and transmits the data from a source to a destination. Each I/O agent performs the I/O operations according to the I/O type. Multiple I/O agents interface local disks, NASD, or peripheral devices. The agents receive the I/O command from the mapper or block mover. The agent performs low-level I/O operations under the control of the local system call.

The user-level agents and functions can be written as Java processes, which facilitates the porting to various host platforms. The system call for remote-disk access requires changing the page-fault mapping table in the kernel. In an experimental cluster at USC, we modify the device-relevant system calls in Linux to run on Pentium-based PC hosts. The mapper, the mover, and various agent routines form the SIOS middleware package. We aim to make the SIOS package portable to all major OS platforms.

Clusters of workstations or PC clusters are bound to become commodity products in the computer industry. However, the major difficulty in clustering lies in the lack of adequate SSI software support. To build multicomputer clusters with SSI, efficient mechanisms or protection schemes are needed for global interprocess communication and for global security control without access conflicts or unauthorized access of shared resources in the cluster.

We need a dynamic mechanism to effectively manage the fast-changing cluster environment. The need for load balancing and process migration add another dimension to the challenge in clustering multiple computers. In the industrial track, Wolfpack for Intel-based Windows NT servers, Berkeley NOW, and Solaris MC for Unix workstations are all aimed at high availability, scalability, and manageability.

The rapid growth in multimedia and WWW applications has further increased the demands on clustered and network-based platforms. These applications demand higher computing power and communication bandwidth, more flexible control of the cluster resources, and higher availability and fault tolerance. SSI clusters or robust clusters will efficiently meet these requirements.

Java-based intelligent agents are also suitable for distributed cluster computing. This applies especially to distributed financial computing, information retrieval in digital libraries, and electronic business. Furthermore, PC or workstation clusters demonstrate potential for large-scale database-search or data-mining applications. Robust clusters will be more cost-effective for bioinformatics, telemedicine, telemarketing, data warehousing, and distance learning than are today's mainframes or supercomputers.

Acknowledgments

This research was jointly carried out at the Internet and Cluster Computing Research Laboratory of the University of Southern California and at the High-Performance Computing Research Laboratory at the University of Hong Kong. The research was supported partially by Hong Kong RGC Grants HKU 2/96C, HKU 7022/97E, and HKU 7032/98E, by the development fund of the HKU Area of Excellence in Information Technology, and by a research grant from the USC School of Engineering.

References

1. K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, WCB/McGraw-Hill, New York, 1998.
2. Y.A. Khalidi et al., "Solaris MC: A Multicomputer Operating System," *Proc. USENIX 1996 Ann. Tech. Conf.*, Usenix Assoc., Berkeley, Calif., 1996, pp. 191–203.
3. C. Amza et al., "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, Vol. 29, No. 2, Feb. 1996, pp. 18–28.
4. T.E. Anderson et al., "The Interaction of Architecture and Operating System Design," *Proc. Fourth Int'l Conf. Architectural Support Programming Languages and OS*, ACM Press, New York, 1991, pp. 108–120.
5. S. Zhou, "LSF: Load Sharing and Batch Queuing Software," Platform Computing Corp., North York, Canada, 1996.
6. M. Stonebraker and G.A. Schloss, "Distributed RAID: A New Multiple Copy Algorithm," *Proc. Sixth Int'l Conf. Data Eng.*, 1990, pp. 430–443.
7. G.A. Gibson et al., "File Server Scaling with Network-Attached Secure Disks," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems,* ACM Press, 1997, pp. 272–284.
8. J.S. Plank, K. Li, and M.A. Puening, "Diskless Checkpointing," *IEEE Trans. Parallel and Distributed Systems*, Vol. 9, No. 10, Oct. 1998, pp. 972–986.
9. N.H. Vaidya, "A Case for Two-Level Distributed Recovery Schemes," *Proc. ACM Int'l Conf. Measurement Modeling Computer Systems*, ACM Press, 1995, pp. 65–73.
10. K. Li, J. Naughton, and J. Plank, "Low-Latency Concurrent Checkpointing for Parallel Programs," *IEEE Trans. Parallel and Distributed Systems,* Vol. 5, No. 8, 1994, pp. 874–879.
11. H. Jin and K. Hwang, *Reconfigurable RAID-5 and Mirroring Architectures for Building High Availability Clusters of Workstations*, tech. report, Internet and Cluster Computing Laboratory, Univ. of Southern California, Los Angeles, Calif., 1999.

Kai Hwang is a professor of electrical engineering and computer science at the University of Southern California. An IEEE Fellow, he specializes in computer architecture, digital arithmetic, and parallel processing. He is the founding Editor-in-Chief of the *Journal of Parallel and Distributed Computing*. He received the 1996 Outstanding Achievement Award at the International Symposium on Parallel and Distributed Processing Techniques and Applications. His research interests lie in network-based multicomputer architecture and middleware development for availability and SSI in cluster applications. He earned his PhD in electrical engineering and computer science from the University of California, Berkeley. Contact him at the Dept. of Electrical Eng. Systems, EEB Rm.106, Univ. of Southern California, Los Angeles, CA 90089; kaihwang@usc.edu.

Hai Jin is an associate professor of computer science at Huazhong University of Science and Technology in China. He has worked at the University of Hong Kong, where he participated in the HKU Cluster project. His research interests cover parallel I/O, RAID architecture design, and cluster benchmark experiments. He received his PhD in computer engineering from the Huazhong University of Science and Technology. Presently, he works as a visiting scholar at the Internet and Cluster Computing Laboratory at the University of Southern California. Contact him at hjin@ceng.usc.edu.

Edward Chow is working on his MPhil degree at the University of Hong Kong's computer engineering program. His research interest lies mainly in fault-tolerant computer architecture design and evaluation. He received his BS in electrical and electronic engineering from the University of Hong Kong. Contact him at cychow@eee.hku.hk.

Cho-Li Wang is an assistant professor of computer science at the University of Hong Kong. His research interests include high-speed networking, computer architectures, and software environments for distributed multimedia applications. He received his BS in computer science and information engineering from National Taiwan University and his MS and PhD in computer engineering from the University of Southern California. Contact him at clwang@csis.hku.hk.

Zhiwei Xu is a professor and the chief architect at the National Center for Intelligent Computing Systems, Chinese Academy of Sciences, Beijing. He leads a design group at NCIC building a series of cluster-based superservers and scalable multicomputers. His research interests lies mainly in network-based cluster computers, parallel programming, and software environments. He received his PhD in computer engineering from the University of Southern California. Contact him at zxu@apple.ncic.ac.cn.