Ausarbeitung
über das Thema

# The Network RamDisk

## Using Remote Memory on Heterogeneous NOWs

**von**
**Svetoslav Vasilev**

# Inhaltsverzeichnis

# 1. Einleitung

***

Effiziente und zuverlässige Dateispeicherung ist von großer Bedeutung für die moderne Computerindustrie. Das am häufigsten verwendete Gerät in den meisten Computersystemen ist die traditionelle Festplatte. Sie gewährleistet eine relativ hohe Integrität der Daten. Das größte Problem aber ist die Latenzzeit, die für die Festplattenzugriffe charakteristisch ist. Das Problem wird noch größer, wenn wir die Latenz vom Standpunkt des Prozessors betrachten. Die Taktfrequenz des Prozessors erhöht sich um ein Vielfaches schneller als sich die Festplattenlatenz verringert. Das heißt, dass der Preis der Verwendung einer Festplatte, in Prozessorzyklen gemessen, ständig steigt. Als Folge davon würden die Anwendungen, die bezüglich der Latenz sehr empfindlich sind, darunter leiden, wenn sie ihre Daten auf der Festplatte speichern müssten. Dazu gehören: Transaktionsbasierte Systeme, Visualisierungssysteme, Webserver, Webproxies, Compiler, etc.

Einige Dateisysteme versuchen ihren Datenzugriff zu beschleunigen, indem sie einen Anteil des Fileserverspeichers als Festplattencache benutzen. Natürlich ist aber dieser Prozess bis zur Größe des Systemspeichers begrenzt.

Die neue Konzeption, die hier beschrieben wird, greift das Problem der hohen Latenz auf eine sehr günstige Art auf. Wir werden sie im weiteren Verlauf als Network RamDisk bezeichnen. Man weiß, dass die heutzutage verbreiteten Netzwerktechnologien eine I/O-Latenzzeit im Mikrosekundenbereich leisten können und eine große Bandbreite bieten. Die Idee ist folgende, anstatt eine traditionelle Festplatte zu benutzen, werden wir die andernfalls nicht benutzten Hauptspeicher einiger vernetzter Computer als ein gemeinsames schnelles Speichergerät verwenden. Auf solche Art erreichen wir eine minimale Latenz für die Lese- und Schreiboperationen. Der zweite sehr wichtige Punkt ist die Zuverlässigkeit der Daten, wobei wir Ideen von gut bekanten RAID-Technologien benutzen werden. Zusätzlich werden wir noch Anforderungen bezüglich Portabilität und Transparenz stellen.

# 2. Das Design von Network RamDisk

***

Die Network RamDisk (NRD) ist ein verteiltes System mit niedriger Latenz. Sie besteht aus einer Maschine, die als NRD-Client dient und aus einer Anzahl von Workstations, die als NRD-Server funktionieren.

## 2.1 Server und Client Workstations

Die Server sind die Workstations, die einen vorbestimmten Anteil ihres Speichers durch NRD ausnutzen lassen. Die Workstations, die den auf solche Art erzeugten Speicher nutzen, werden Clients genannt. Alle Workstations können gleichzeitig als Server und Client arbeiten. Die NRD funktioniert als Treiber, der mit dem Betriebssystem des Clients verknüpft werden muss. Natürlich muss der Client auch eine Netzwerkverbindung zu den Servern haben. Die NRD dient als normale Festplatte, die auf der Clientmaschine montiert ist und jeder Rechner im Netzwerk kann auf sie durch den Mapping-Prozess zugreifen.

## 2.2 Device Operation

Der NRD-Serverprozess läuft auf allen Serverworkstations, speichert Dateiblöcke und bearbeitet alle Lese- und Schreibanforderungen, die blockweise ausgeführt werden. Der NRD-Client nimmt die I/O-Anforderungen an wie ein normales Blockgerät und wenn er Blöcke lesen oder schreiben will, durchsucht er seine Blocktabelle, um herauszufinden, auf welchem Server sich der entsprechende Block befindet und beginnt diesem Server Anforderungen zu senden.

Normalerweise befinden sich alle Dateien von NRD in den Serverhauptspeichern. Es ist aber möglich, dass ein Native-Prozess wie Memoryswapping auf dem Server gestartet wird, was einen Teil des Speichers auf die Festplatte auslagern würde. Dieses Problem könnte gelöst werden, indem der Server dem Client eine Nachricht sendet, in der er ihn über seine Speicherauslastung informiert. Der Client versucht einen anderen Server mit genug freiem Speicher zu finden und einige Blöcke von dem belasteten Server zu übertragen. Falls das unmöglich ist, ist eine Verzögerung nicht zu vermeiden.

## 2.3 Zuverlässigkeitsanforderungen

In einem verteilten System könnte eine Workstation zu beliebiger Zeit ausfallen. Am häufigsten passiert dies wegen Softwarefehlern. Falls das ausgefallene System als Client funktioniert, kann es, nachdem es neu gestartet wurde, seine Dateien wiederherstellen, indem es sich zu den NRD-Servern verbindet und das `fsck`-Kommando ausführt, womit alle Uneinheitlichkeiten korrigiert werden. Falls aber die ausgefallene Maschine als ein NRD-Server funktioniert, würde sie die Clientdatei verlieren. Es ist klar, dass es inakzeptabel ist, wenn eine Anwendung, die auf dem Client läuft, wegen des Ausfalls einer Workstation seine Datei verliert. Stattdessen wollen wir in der Lage sein, die Dateiblöcke wiederherzustellen.

Um die Daten nicht zu verlieren, schreiben einige Systeme sie auf die Festplatte. Anstatt so zu verfahren, ist NRD so konstruiert, dass sie die verlorenen Dateien wiederherstellen kann. Um dieses Ziel erreichen zu können, müssen einige Redundanzformen eingeführt werden, wobei aber die folgenden Bedingungen in Angriff genommen werden müssen.

1. Das runtime overhead sollte minimal sein, weil das ein Preis ist, der immer bezahlt wird, auch wenn es keine Serverausfälle gibt.
2. Das memory overhead sollte so klein wie möglich sein, weil der Speicher, der für die Zuverlässigkeit reserviert wird, für memory pages benutzt werden könnte.
3. Das runtime overhead der Wiederherstellung sollte niedrig sein.

Zwei Zuverlässigkeitsstrategien (Speicherstrategien) werden beschrieben: **Mirroring** und **Parity Logging**.

# 2.4 Speicherstrategien

## 2.4.1 Mirroring

Das ist die einfachste Verfahrensweise, mit der Zuverlässigkeit erreicht werden kann. Man braucht nur die Daten auf einen verschiedenen Speicher zu kopieren. Diese Redundanzmethode ist sehr einfach zu implementieren und bietet eine hohe Wiederherstellungszuverlässigkeit. Im Falle, dass ein Server ausfällt, braucht der Client nur den entsprechenden Server kennen, der die Spiegeldaten noch gespeichert hat. Es gibt bei dieser Methode nur zwei Nachteile, diese sind aber sehr gewichtig:

a) Die Hälfte des Speichers wird verschwendet

b) Das runtime overhead wird verdoppelt, weil jede Datei zweimal gesendet werden muss

## 2.4.2 Parity Logging

Vor dem Blick auf die Funktionsweise von ´´Parity Logging``, muss das Konzept von ´´Parity Group`` definiert werden. Eine Parity-Gruppe ist eine Anzahl von Dateiblöcken und ein entsprechender Parity-Block, der von XORen von allen Dateiblöcken erzeugt wird. Falls einer von den Dateiblöcken verloren ginge, könnten wir ihn wiederherstellen, wenn wir die anderen Dateiblöcke in der Parity-Gruppe XORen.

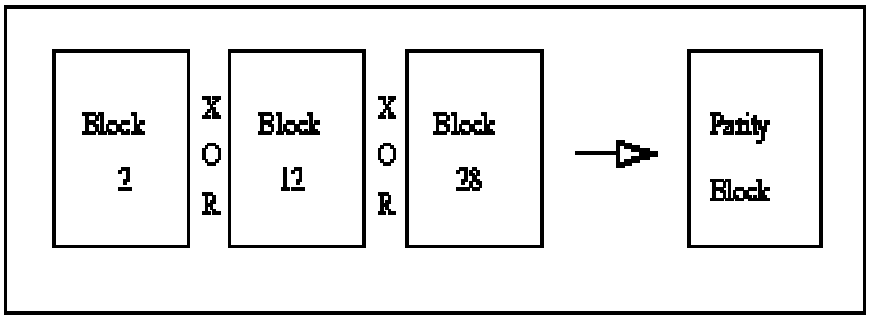



**Abb. 1. Parity Group**

Damit wir Zuverlässigkeit garantieren können, soll jeder Block von der Parity-Gruppe auf verschiedenen Servern abgespeichert werden.

Betrachten wir jetzt wie das ´´Parity Logging`` funktioniert. Angenommen der Client benutzt *S* Server – einen Parityserver und *S-1* Speicherserver. Beim Client befindet sich ein Puffer (ein Block), der am Anfang mit Nullen initiiert wird und mit dem alle Dateiblöcke, die zu übertragen sind, geXORt werden. Wenn alle *S-1* Blöcke verschoben sind, wird auch der Puffer in den Parityserver verschoben und das bedeutet, dass das Erstellen der Parity-Gruppe beendet ist. Wenn man diese Technik benutzt, hat man ein minimales runtime overhead, weil für jeden Block nur ***1 + 1/S*** Transfers gemacht werden müssen. Falls ein Server ausfällt, können die fehlende Dateien mit Hilfe von XORen der entsprechenden Parity-Gruppe wiederherstellt werden.

Die ganze Information über den Verbleib jedes Blocks und die Parity-Gruppe, dazu er gehört, befinden sich auf der Clientmaschine. Immer wenn ein Block überschrieben werden muss, wird er in der alten Parity-Gruppe als nicht aktiv markiert. Nur wenn alle Blöcke einer Parity-Gruppe als nicht aktiv markiert sind, könnte der von ihnen besetzte Speicher wieder benutz werden. Diese Bedingung ist wegen der Zuverlässigkeitsanforderungen gesetz werden und nähmlich, dass die Parity-Gruppe überschrieben werden kann, nur wenn alle ihre Blöcke als nicht aktiv markiert sind, weil zur jeden Zeit alle Dateien sollen

wiederherstellt werden können. Deswegen ist es klar, dass jeder Server Extraspeicher haben muss, um verschiedene Versionen von ein- und demselben Block gleichzeitig speichern zu können. Daher, im Falle des Speichermangels, muss einer der Server Garbage-Collection durchführen.

### 2.4.3 Parity Caching

Auf alle Fälle könnte das ´´Parity Logging``-Verfahren nicht benutzt werden, ohne zu einem bedeutsamen runtime overhead zu kommen. Das kommt daher, dass bei jeder Dateianforderung ein Netzwerkzugriff gemacht werden muss. Da bei jeder Datenanforderung das Filesystem verschiedene Menge von Blöcken sendet, in der Regel 1 bis 8, ist die Latenzzeit des ganzen System gleich der Latenz des Netzwerks mal die Anzahl der Anforderungen.

Daher ist das neue zuverlässige Verfahren entwickelt worden, das die Latenzengstelle für die I/O-Anforderungen zu verringert hat. Diese Methode, die **Parity Caching** genannt wird, verwendet eine kleine Menge des Speichers des NRD-Clients als Parity-Cache-Speicher. Der Parity-Cache sammelt fixierte Mengen von Blöcken, was der Parity-Caching-Methode die Gelegenheit gibt, mehrere Blöcke als ´´Memory-Stripping´´-Einheit zu benutzen. Offensichtlich hat das Parity-Caching-Verfahren eine kleinere Latenz als das einfache Parity-Logging-Verfahren.

Nehmen wir an, dass ein Client und *S* Server existieren. Wir brauchen einen Server, der die Parity-Blöcke speichert – Parityserver und die andere *S-1* Server speichern die normalen Dateiblöcke, sie werden Storagerserver genannt. Dementsprechend hat der Client *S-1* Speicherpuffer und ein Paritypuffer. Wenn wir die Plattengeometrie des Laufwerks betrachten, können wir feststellen, dass es bequem wird, die maximale Anzahl von Blöcken pro Anforderung auf *X* zu setzen. Auf diese Art werden alle Anforderungen bis *X* Blöcke begrenzt und die Parity-Cache-Größe wird *X***S* Blöcke (1 Block = 512 Bytes). Jeder Puffer im Parity-Cache entspricht genau einem Server, was bedeutet, dass die Dateiblöcke die in diesem Cache abgespeichert sind, dem entsprechenden Server gesendet werden (Abb. 2).

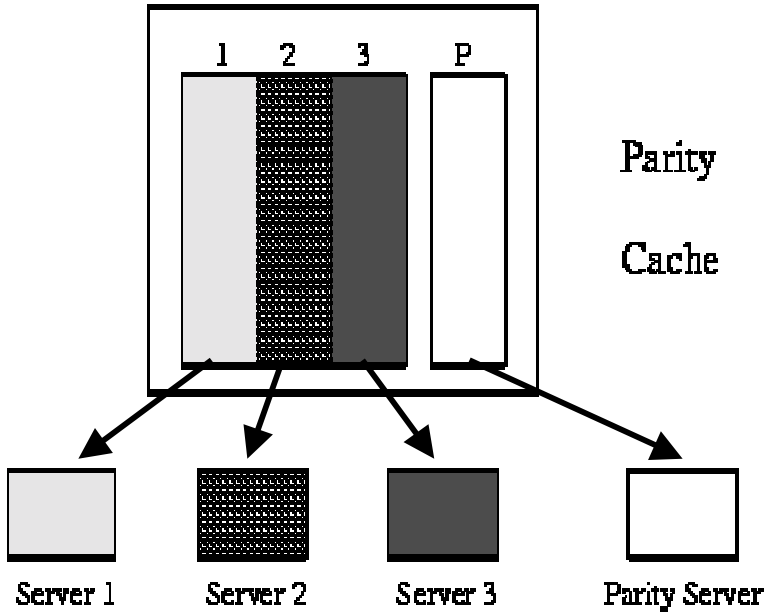



Abb. 2 der Parity Cache

Wenn eine Schreibanforderung in Größe von X Blöcken entsteht, speichert der Client die X Blöcke in einen Speicherpuffer ab und aktualisiert die Tabelle die den Speicherplatz jedes Blocks zeigt. Wenn alle S-1 Speicherpuffer im Cache voll sind, berechnet der Client den Parity-Block im Parity-Puffer, sendet die ganzen Daten den entsprechenden Servern und entleert den Cache.

### 2.4.4 Adaptive Parity Caching

Ein anderes Problem ist die Existenz von ungültigen Blöcken, und das Entfernen von diesen. Wegen der Widerherstellbarkeit wird jeder Block, der überschrieben wird, als ungültig markiert und kann nicht entfernt werden, bis alle Blöcke in seiner Parity-Gruppe als ungültig markiert wurden. Und da kein Block einen fixierten Platz hat, kommt es zu memory und runtime overhead.

Aufgrund unserer Kenntnisse von Plattenlaufwerken und Dateisystemen vermuten wir, dass das Filesystem solche Blöcke abspeichern würde, die entweder zur selben Datei gehören, oder die Teil eines einzelnen Dateistroms auf der sequentiellen Speicherstelle sind. Das alles bedeutet, dass die Dateiblocknummern benachbart werden. Es ist auch höchstwahrscheinlich, dass diese Blöcke zusammen gelesen, geschrieben oder überschrieben würden. Falls die zu überschreibenden Blöcke zusammen abgespeichert würden, würde das die Entfernung automatisch bewirken. Anhand dieser Beobachtung erweitern wir das ´´Parity Caching`` bis zum ´´**Adaptive Parity Caching`` (APC)**. Die Hauptfunktion der APC-Methode ist zu versuchen die Dateiblockströme, die das Dateisystem dem NRD-Treiber sendet, zu entschachteln.

Bei dem APC-Verfahren benutzen wir *P* Parity-Caches, wie oben erwähnt. Wir nennen jeden Cache Stream-Cache, weil er einen einzelnen Dateistrom zu erfassen versucht (Abb. 3).

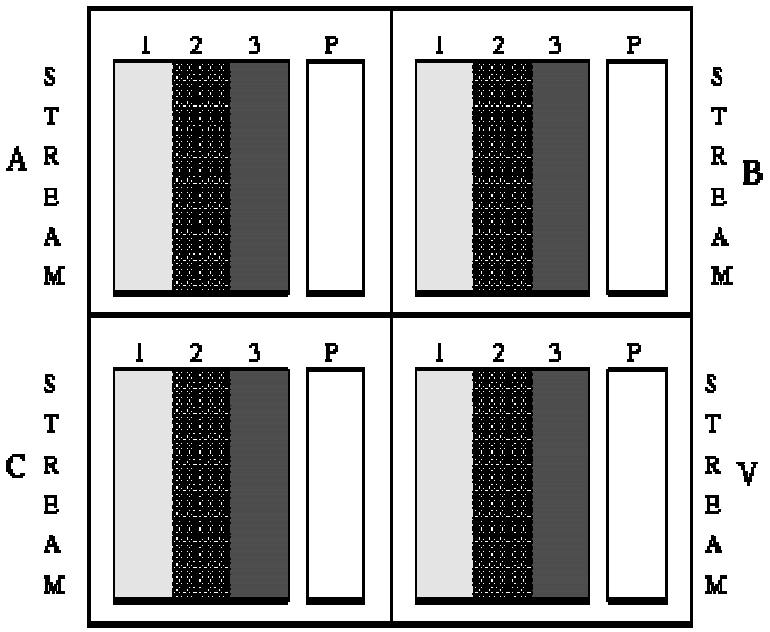



Abb. 3 Adaptive Parity Caching

Auf der Abbildung sieht man 3 Stream-Caches und einen Victim-Cache. Wenn der Client den Stream-Cache auswählen soll, um die *X* Blöcke abzuspeichern, verwendet er das folgende Verfahren:

1. Falls alle Stream-Caches leer sind, speichere die Blöcke in den ersten Stream-Cache ab.

2. Falls es einen nicht leeren Stream-Cache gibt:

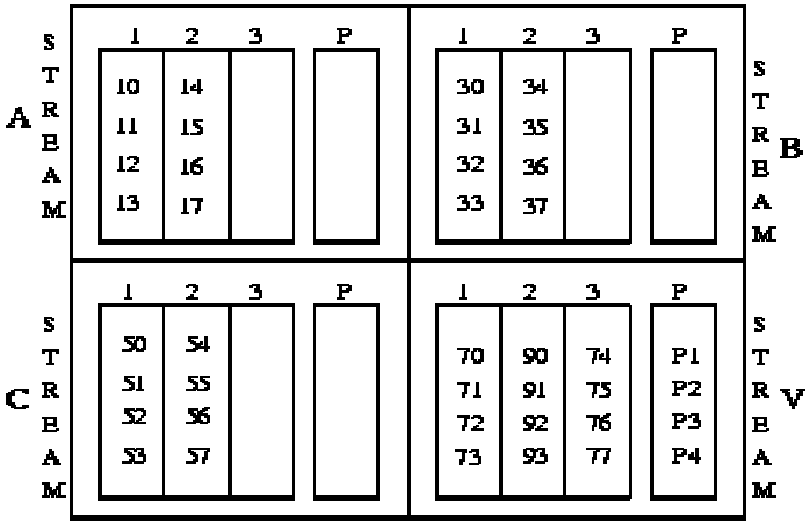
   a) Suche den nicht leeren Cache durch und wenn die X Blöcke benachbart sind, speichere sie in den ersten freien Puffer desselben Caches ab.

   b) Falls es einen leeren Cache gibt, speichere die Blöcke in seinem ersten Puffer ab.

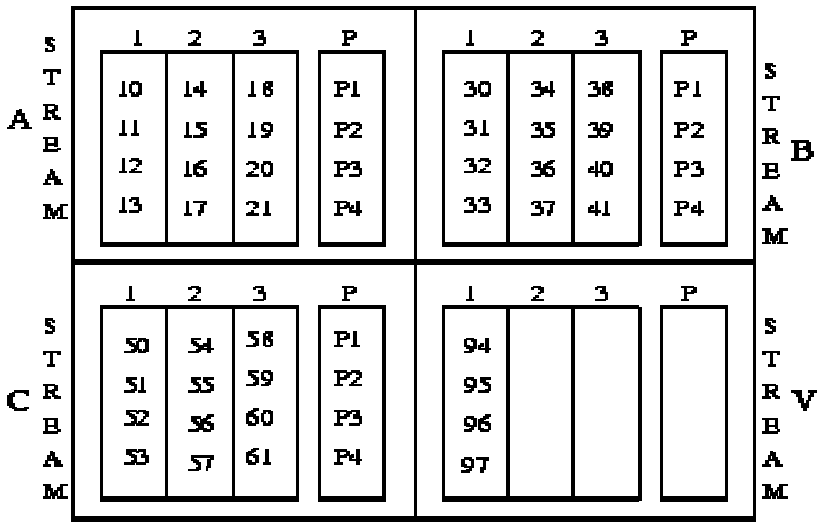
   c) Falls es keine Übereinstimmung mit dem Obenerwähnten gibt, speichere die Blöcke in den ersten freien Puffer des Victim-Caches ab.

Im Falle, dass wir auf nicht benachbarte Dateiblöcke warten, gibt es eine Stream-Cache-Zeitüberschreitung (Timeout). Falls der Victim-Cache zweimal voll war und einige Caches noch teilweise frei waren, füllt der Client die leer gebliebenen Caches mit Nullen und arbeitet die vollen Caches ab.

Beispiel: Der Dateiblockstrom ist (10, 11, 12, 13), (14, 15, 16, 17), (30, 31, 32, 33), (34, 35, 36, 37), (50, 51, 52, 53), (70,71,72,73), (90,91,92,93), (34,35,36,37), (14,15,16,17), (54,25,26,27), und (74,75,76,77). Das Adaptive Parity Caching verteilt die benachbarten Datenblöcke in einzelne Caches. Die Blöcke mit den Nummern 70 und größer sind im Victim-Caches abgespeichert, weil keine anderen freien Stream-Caches gibt.




Auf der nächsten Abbildung verfahren die Dinge auf dieselbe Art. Der neue Datenstrom ist: (18,19,20,21), (94,95,96,97), (38,39,40,41), (58,59,60,61).



Auf solche Art alle Stream-Caches sind voll und werden den entsprechenden Servern gesendet.

# 4. Implementierung

***

Die Network RamDisk besteht aus einem Client, der blockweise Lese- und Schreibeanforderungen produziert und aus einigen Servern, die diese Anforderungen erfüllen. Voll funktionierende Network RamDisk Clientsysteme sind in Digital Unix 4.0 und in Linux 2.0.33 eingebaut und Serversysteme sind in Digital Unix 4.0 und Solaris 2.5 eingebaut.

## 4.1 Der Network RamDisk Client

Wenn der Client gestartet wird, macht er die Systemeinstellung. Er stellt fest wie viele NRD-Server im Netz zur Verfügung stehen, wie viel Speicher wird von diesen Servern überlassen. Der Client „vereinbart“ mit den NRD-Servern die Zuverlässigkeitsstrategie, dann erzeugt er die Datenstrukturen und initialisiert sie. Nach diesem Prozess verbindet sich der Client mit den Servern, bearbeitet die I/O-Anforderungen und ersetzt vollständig die Festplatte. Das Betriebsystem weiß überhaupt nicht, dass die Dateien nicht auf der Festplatte abgespeichert werden.

## 4.2 Der Network RamDisk Server

Die NRD-Server haben die Aufgabe den Sockets zu lauschen und wenn eine Lese- oder Schreibanforderung kommt entsprechend zu reagieren, indem sie Dateiblöcke lesen oder schreiben. Im Falle dass der Speicher eines Servers ausgelastet ist, soll dieser dem Client eine Nachricht schicken, damit er einen anderen Server findet.