Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Τεχνικές Εισόδου/Εξόδου και Χρονοδρομολόγησης για την Αποδοτική Χρήση Μοιραζόμενων Αρχιτεκτονικών Πόρων σε Συστοιχίες Κόμβων Συμμετρικής Πολυεπεξεργασίας

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Ευάγγελος Λ. Κούκης

Αθήνα, Ιανουάριος 2010

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Τεχνικές Εισόδου/Εξόδου και Χρονοδρομολόγησης για την Αποδοτική Χρήση Μοιραζόμενων Αρχιτεκτονικών Πόρων σε Συστοιχίες Κόμβων Συμμετρικής Πολυεπεξεργασίας

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Ευάγγελος Λ. Κούκης

Συμβουλευτική Επιτροπή: Νεκτάριος Κοζύρης
Γεώργιος Παπακωνσταντίνου
Παναγιώτης Τσανάκας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 29η Ιανουαρίου 2010.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Γεώργιος Παπακωνσταντίνου	Παναγιώτης Τσανάκας
Αναπ. Καθηγητής ΕΜΠ	Ομ. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ
. . . . .	. . . . .	. . . . .
Ανδρέας-Γεώργιος	Άγγελος Μπίλας	Νικόλαος Παπασπύρου
Σταφυλοπάτης	Αναπ. Καθηγητής	Επίκ. Καθηγητής ΕΜΠ
Καθηγητής ΕΜΠ	Πανεπιστημίου Κρήτης	
	. . . . .	
	Διομήδης Σπινέλλης	
	Καθηγητής ΟΠΑ	

Αθήνα, Ιανουάριος 2010

. . . . . .

Ευάγγελος Λ. Κούκης
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Η παρούσα διατριβή χρηματοδοτήθηκε από το έργο ΠΕΝΕΔ 2003, που υλοποιείται στο πλαίσιο του Μέτρου 8.3 του Ε.Π. Ανταγωνιστικότητα Γ' Κοινοτικό Πλαίσιο Στήριξης και συγχρηματοδοτείται κατά: 80% της Δημόσιας Δαπάνης από την Ευρωπαϊκή Ένωση - Ευρωπαϊκό Κοινωνικό Ταμείο, 20% της Δημόσιας Δαπάνης από το Ελληνικό Δημόσιο - Υπουργείο Ανάπτυξης - Γενική Γραμματεία Έρευνας και Τεχνολογίας.

στους γονείς μου
Λεωνίδα και Παναγιώτα
στους αδερφούς μου
Χρήστο, Παναγιώτη και Νίκο

What we call the beginning is often the end.
And to make an end is to make a beginning.
The end is where we start from.

— T.S. Eliot

Έκδοση 61, 2010-02-02 09:36:42Z από vangelis.

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.9995.

Χρησιμοποιήθηκαν οι γραμματοσειρές Minion Pro και Consolas.

Contents

Αντί Προλόγου **xi**

Abstract **xiii**

Περίληψη **1**

0.1 Εισαγωγή 2
0.2 Υπόβαθρο 8
0.3 Σχεδίαση και υλοποίηση του gmblock 16
0.4 Συγχρονισμένες λειτουργίες αποστολής GM 32
0.5 Σχεδίαση πελάτη και αποτίμηση από άκρο σε άκρο 41
0.6 Σχετικές εργασίες 54
0.7 Συμπεράσματα και μελλοντικές κατευθύνσεις 57

1 Introduction **61**

1.1 Motivation 61
1.2 Contribution 65
1.3 Outline 68

2 Background **69**

2.1 nbd systems and applications 69
2.2 User level networking 74
2.2.1 Basic concepts 74
2.2.2 Implementation in Myrinet/GM 80
2.3 The Linux Block Layer 86
2.3.1 Main functionality 86
2.3.2 Servicing a block I/O request 88

3 Design and implementation of gmblock **93**
3.1 Design of gmblock's nbd mechanism 93
3.1.1 Traditional nbd designs 93
3.1.2 GMBlock: An alternative data path with memory bypass . . . 99
3.2 Implementation details 101
3.2.1 GM support for buffers in Lanai SRAM 102
3.2.2 Linux VM support for direct I/O with PCI ranges 104
3.3 Experimental evaluation 104
3.3.1 Experiment 1a: Local disk performance 106
3.3.2 Experiment 1b: Remote read performance 107
3.3.3 Experiment 1c: Effect on local computation 112
3.3.4 Experiment 1d: Remote write performance 112
3.4 Discussion 114
3.4.1 Sharing of structured data 114
3.4.2 Caching and prefetching 115
3.4.3 Applicability to other interconnection technologies 117
3.4.4 Applicability to low-power designs 118

4 Synchronized GM send operations **119**
4.1 Motivation 119
4.2 Design 120
4.3 Implementation issues on Myrinet/GM 123
4.4 Experimental evaluation 125
4.4.1 Experiment 2a: Synchronized Send Operations 127
4.4.2 Experiment 2b: RAID data movement 127

5 Client-side issues and end-to-end evaluation **131**
5.1 Design considerations for a kernelspace nbd client 131
5.1.1 Principles of operation 131
5.1.2 Client-side data movement 132
5.2 Proposed extensions to Myrinet/GM 137
5.2.1 Design 137
5.2.2 Implementation of GM scatter-gather operations 139
5.3 Kernelspace gmblock client 140
5.4 Parallel filesystem deployment over gmblock 141
5.4.1 Experiment 3a: Single-node IOzone performance 143
5.4.2 Experiment 3b: Multiple-node IOzone performance 146
5.4.3 Experiment 3c: Server workload 147
5.4.4 Experiment 3d: MPI-IO Application 148

6 Related work **151**

7 Conclusions and future directions **161**

Bibliography **165**

Publications **177**

Αντί Προλόγου

Το κείμενο του διδακτορικού είναι σχεδόν έτοιμο. Κάθομαι μπροστά στον υπολογιστή, έχω ένα άσπρο παράθυρο του Νόουτπαντ μπροστά μου, σκέφτομαι πώς να ξεκινήσω να γράφω τον πρόλογο και το μυαλό μου πλημμυρίζει πρόσωπα.

Φτάνοντας στην ολοκλήρωση του διδακτορικού, βλέπω ότι αυτό απαιτεί αρκετό χρόνο, πολύ κόπο και *ανθρώπους*. Θα ήθελα λοιπόν στο σημείο αυτό να εκφράσω τις ευχαριστίες μου σε ένα πλήθος ανθρώπων που συνέβαλαν, ο καθένας με τον τρόπο του, στην ολοκλήρωση της εργασίας αυτής.

Κατά τη διάρκεια των μεταπτυχιακών σπουδών μου, είχα την τύχη να έχω συναδέλφους και πραγματικούς φίλους στο εργαστήριο μερικούς από τους πιο ενδιαφέροντες, ζωντανούς ανθρώπους που έχω γνωρίσει ποτέ. Τους ευχαριστώ για την καθημερινή αλληλεπίδραση, τις συναρπαστικές συζητήσεις τόσο εντός όσο και εκτός εργαστηρίου, για τεχνικά και όχι τόσο τεχνικά θέματα. Θα ήταν μάταιο να προσπαθήσω να τους αναφέρω έναν-έναν, τους ευχαριστώ μέσα από την καρδιά μου και εύχομαι καλή επιτυχία στα τωρινά και μελλοντικά τους σχέδια.

Θα ήθελα να ευχαριστήσω την επιτροπή παρακολούθησης του διδακτορικού, πρώτα τον επιβλέποντα καθηγητή κ. Νεκτάριο Κοζύρη, για την επιστημονική καθοδήγησή του, την ομαλή συνεργασία μας όλα αυτά τα χρόνια και την ευκαιρία που μου έδωσε να αποκτήσω πολύτιμη εμπειρία με συναρπαστικές τεχνολογίες κατά τη διάρκεια της έρευνάς μου στο Εργαστήριο Υπολογιστκών Συστημάτων. Επίσης, θα ήθελα να ευχαριστήσω ιδιαίτερα τον κ. Άγγελο Μπίλα, αναπ. καθηγητή στο Πανεπιστήμιο Κρήτης για τη διάθεσή του να μοιραστεί σε συζητήσεις τις εκτιμήσεις του για την εξέλιξη των

υπολογιστικών συστημάτων στο μέλλον και να συνεισφέρει πολύ χρήσιμα σχόλια στη δουλειά μου. Τέλος, ευχαριστώ τον καθηγητή κ. Γεώργιο Παπακωνσταντίνου, η παρουσία του οποίου υπήρξε ηθικό πρότυπο και πηγή έμπνευσης για εμένα, όλα αυτά τα χρόνια στο εργαστήριο.

Η παρούσα διατριβή θα ήταν σίγουρα πολύ φτωχότερη χωρίς τις μακροσκελείς τεχνικές συζητήσεις με τους Υ.Δ. Γιώργο Τσουκαλά και Κορνήλιο Κούρτη, καθώς και το Δρα Άρη Σωτηρόπουλο. Η τεχνική τους κατάρτιση σε συνδυασμό με την προθυμία τους να με προσγειώνουν απότομα στην πραγματικότητα με εύστοχη, καλοπροαίρετη κριτική συνέβαλαν αποφασιστικά στην ολοκλήρωση της δουλειάς αυτής.

Θα ήθελα επίσης να αναφερθώ ειδικά στο Δρα Γιώργο Γκούμα, για την ενθάρρυνσή του, την πίστη του στην προσπάθειά μου, την αξιοθαύμαστη ικανότητά του να εμπνέει αυτοπεποίθηση σε δύσκολες στιγμές και να κάνει τα πράγματα να φαίνονται απλούστερα. Ο Γιώργος είχε την υπομονή να διαβάσει πολλές αρχικές εκδόσεις της διατριβής και να προτείνει αλλαγές που βελτίωσαν καθοριστικά τη δομή και το περιεχόμενό της. Προφανώς, τυχόν λάθη και παραλείψεις αποτελούν αποκλειστικά δική μου ευθύνη.

Ευχαριστώ επίσης τους φίλους μου από τα προπτυχιακά χρόνια και λίγο αργότερα, το Λευτέρη, το Θάνο, τη Μαρίνα, το Φώτη, το Χρήστο, για τις συζητήσεις, τα γέλια, τις βόλτες, το χρόνο που περάσαμε μαζί όλα αυτά τα χρόνια.

Τελευταίοι, αλλά σημαντικότεροι, η οικογένειά μου: η μητέρα μου, Παναγιώτα, ο πατέρας μου, Λεωνίδας, ο Χρήστος, ο Παναγιώτης, ο Νίκος. Προφανώς δεν μπορώ να χωρέσω σε πέντε γραμμές την αγάπη μου γι'αυτούς, την ευγνωμοσύνη μου για τη βοήθειά τους, την υλική και ηθική στήριξή τους, την υπομονή τους στη – συχνή – γκρίνια μου. Σε αυτούς θα ήθελα να αφιερώσω την παρούσα διατριβή.

Ευχαριστώ.

Βαγγέλης Κούκης
Ιανουάριος 2010

Abstract

Clusters have become prevalent as a cost-effective solution for building scalable parallel platforms to power diverse workloads. Symmetric multiprocessors of multicore chips are commonly used as building blocks for clustered systems, when combined with high-performance interconnection networks, such as Myrinet. SMPs are characterized by resource sharing at multiple levels; Resources being shared include CPU time on cores, levels of the cache hierarchy, bandwidth to main memory, and peripheral bus bandwidth.

The increasing use of clusters for data-intensive workloads, in combination with the trend for ever-increasing cores per processor die, poses significant load on the I/O subsystem. Thus, its performance becomes decisive in determining overall system throughput. To meet the challenge, we need low-overhead mechanisms for transporting large datasets efficiently between compute cores and storage devices. In the case of SMP systems, this means reduced CPU, memory bus and peripheral bus contention.

This work explores the implications of resource contention in SMP nodes used as commodity storage servers. We study data movement in an block-level storage sharing system over Myrinet and find its performance suffers due to memory and peripheral bus saturation. To alleviate the problem, we propose techniques for building efficient data paths between the storage and the network on the server side, and the network and processing cores on the client side. We present gmblock, a system for shared block storage over Myrinet which supports a direct disk-to-NIC server-side data path, bypassing the host CPU and memory bus. To improve handling of large requests and support intra-

request overlapping of network- and disk-I/O with minimal host CPU involvement, we introduce synchronized send operations as extensions to standard Myrinet/GM sends; their semantics support synchronization with an agent external to the NIC, e.g., a storage controller utilizing the direct-to-NIC data path.

On the client side, the proposed system exploits NIC programmability to support protected direct placement of incoming fragments into buffers dispersed in physical memory. This enables end-to-end zero-copy block transfers directly from remote storage to client memory over the peripheral bus and cluster interconnect.

Experimental evaluation of the proposed techniques demonstrates significant increases in remote I/O rate and reduced interference with server-side local computation. A prototype deployment of the OCFS2 shared-disk filesystem over gmblock shows gains for various application benchmarks, provided I/O scheduling can eliminate the disk bottleneck due to concurrent access.

Περίληψη

Οι συστοιχίες (clusters) έχουν επικρατήσει ως οικονομική λύση για την κατασκευή κλιμακούμενων παράλληλων αρχιτεκτονικών, παρέχοντας υπολογιστική ισχύ σε ποικίλες εφαρμογές. Συστήματα Συμμετρικής Πολυεπεξεργασίας (SMPs) από πολυπύρηνους επεξεργαστές χρησιμοποιούνται συχνά ως δομικοί λίθοι στην κατασκευή συστοιχιών, σε συνδυασμό με δίκτυα διασύνδεσης υψηλής επίδοσης, όπως το Myrinet. Τα συστήματα SMP χαρακτηρίζονται από διαμοιρασμό πόρων σε πολλά επίπεδα· στους μοιραζόμενους πόρους περιλαμβάνονται ο χρόνος CPU, επίπεδα της ιεραρχίας κρυφών μνημών, το εύρος ζώνης προς την κύρια μνήμη και το εύρος ζώνης στον περιφερειακό διάδρομο.

Η αυξανόμενη χρήση των συστοιχιών για εφαρμογές απαιτητικές σε δεδομένα, σε συνδυασμό με την τάση για περισσότερους υπολογιστικούς πυρήνες ανά επεξεργαστή, αυξάνει τον φόρτο του υποσυστήματος Εισόδου/Εξόδου. Η επίδοσή του είναι καθοριστική στο συνολικό ρυθμό εξυπηρέτησης του συστήματος. Για το λόγο αυτό, χρειαζόμαστε μηχανισμούς χαμηλής επιβάρυνσης για την αποδοτική μετακίνηση μεγάλων συνόλων δεδομένων ανάμεσα σε υπολογιστικούς πυρήνες και αποθηκευτικά μέσα. Στην περίπτωση των SMP, η απαίτηση αυτή μεταφράζεται σε μειωμένη κατανάλωση χρόνου CPU και εύρους ζώνης στον διάδρομο μνήμης και τον περιφερειακό διάδρομο.

Η παρούσα διατριβή εξερευνά τις επιπτώσεις του ανταγωνισμού για μοιραζόμενους πόρους σε εξυπηρετητές αποθήκευσης. Μελετάμε την κίνηση των δεδομένων σε σύστημα μοιραζόμενης πρόσβασης επιπέδου μπλοκ πάνω από Myrinet και βρίσκουμε ότι ο κορεσμός του διαδρόμου κύριας μνήμης και του περιφερειακού διαδρόμου επιβαρύνει σημαντικά τη λειτουργία του. Για την αντιμετώπιση του προβλήματος, προτείνουμε τεχνικές για την κατασκευή

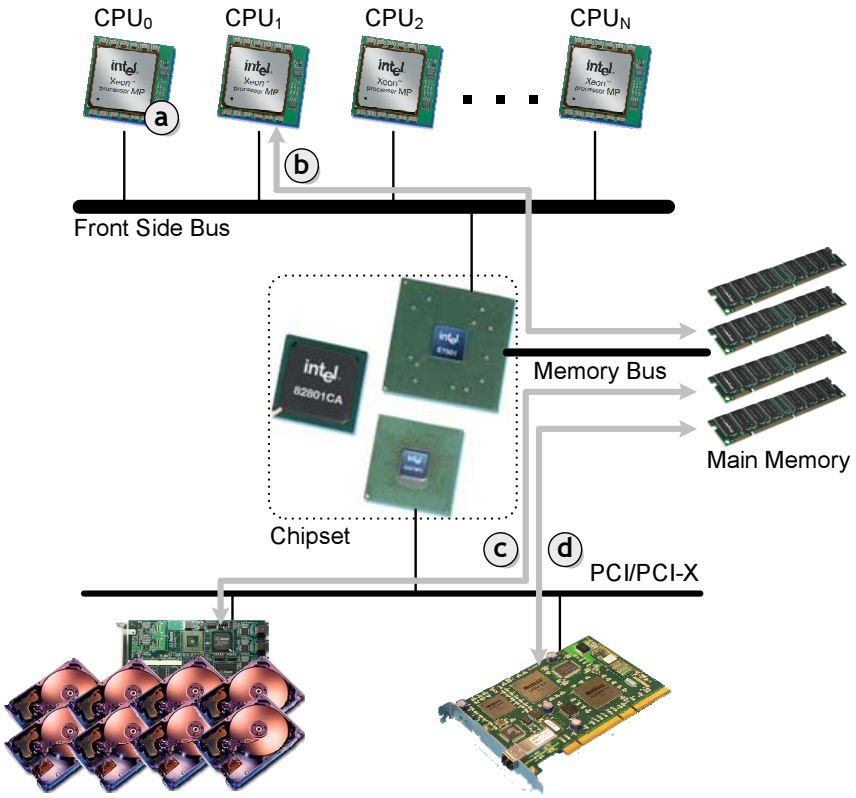
αποδοτικών μονοπατιών δεδομένων ανάμεσα σε αποθηκευτικά μέσα και το δίκτυο, στην πλευρά του εξυπηρετητή, και το δίκτυο και τους υπολογιστικούς πυρήνες, στην πλευρά των πελατών. Παρουσιάζουμε το gmblock, ένα σύστημα μοιραζόμενης πρόσβασης επιπέδου μπλοκ το οποίο υποστηρίζει απευθείας μονοπάτι δεδομένων από το δίσκο σε προσαρμογέα Myrinet, παρακάμπτοντας τον επεξεργαστή και το διάδρομο μνήμης. Για βελτιωμένη υποστήριξη αιτήσεων μεγάλου μήκους και υποστήριξη επικάλυψης των φάσεων ανάγνωσης και δικτυακής αποστολής με ελάχιστη εμπλοκή της CPU, εισάγουμε συγχρονισμένες λειτουργίες αποστολής ως επεκτάσεις στο Myrinet/GM. Η σημασιολογία τους επιτρέπει συγχρονισμό της κάρτας δικτύου με εξωτερικό παράγοντα, π.χ. έναν ελεγκτή αποθήκευσης που χρησιμοποιεί το άμεσο μονοπάτι.

Στην πλευρά του πελάτη, το προτεινόμενο σύστημα εκμεταλλεύεται την προγραμματισιμότητα της κάρτας δικτύου για να υποστηρίξει άμεση τοποθέτηση εισερχόμενων τεμαχίων δεδομένων σε απομονωτές διάσπαρτους στη φυσική μνήμη. Η σχεδίαση αυτή καθιστά δυνατή την κίνηση μπλοκ από άκρο σε άκρο χωρίς αντίγραφα, απευθείας από απομακρυσμένο αποθηκευτικό μέσο στο δίκτυο και τελικά στη μνήμη του πελάτη.

Η πειραματική αποτίμηση των προτεινόμενων τεχνικών δείχνει σημαντική αύξηση του ρυθμού απομακρυσμένης Ε/Ε και μειωμένη παρεμβολή στον τοπικό υπολογισμό στην πλευρά του εξυπηρετητή. Από την εκτέλεση διαφόρων μετροπρογραμμάτων σε εγκατάσταση του παράλληλου συστήματος αρχείων OCFS2 πάνω από το gmblock προκύπτει βελτίωση της απόδοσης του συστήματος, με την προϋπόθεση ότι η χρονοδρομολόγηση Ε/Ε στην πλευρά του εξυπηρετητή εξαλείφει τη στενωπό στους δίσκους λόγω ταυτόχρονης πρόσβασης από πολλούς πελάτες.

0.1 Εισαγωγή

Οι υπολογιστικές συστοιχίες (clusters) χρησιμοποιούνται όλο και συχνότερα για την παροχή υπολογιστικής ισχύος σε πληθώρα εφαρμογών από διάφορα επιστημονικά πεδία. Ως δομικοί λίθοι για την κατασκευή συστοιχιών χρησιμοποιούνται συχνά Συστήματα Συμμετρικής Πολυεπεξεργασίας (SMPs) αποτελούμενα από πολυπύρηνους επεξεργαστές (CMPs). Η απαιτούμενη δικτυακή υποδομή παρέχεται από ένα δίκτυο διασύνδεσης υψηλών επιδόσεων, όπως το Myrinet [BCF+95], το Quadrics [PcFH+01] ή το Infiniband [Inf00].



Σχήμα 1: *Τυπική διάταξη SMP με διάδρομο συστήματος*

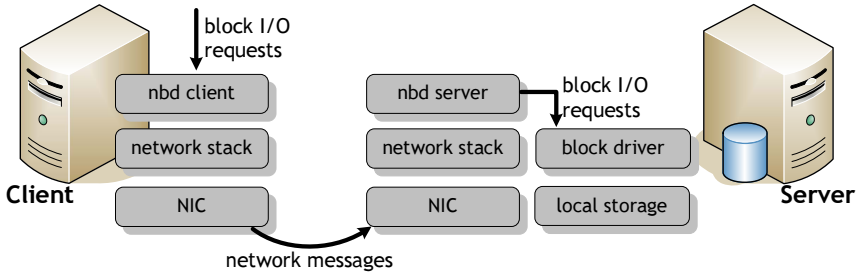
Το Σχ. 1 παρουσιάζει την τυπική δομή ενός κόμβου SMP. Ένας αριθμός επεξεργαστών διασυνδέεται μέσω κοινού διαδρόμου συστήματος (Front-Side Bus) με τον ελεγκτή μνήμης, ο οποίος παρέχει πρόσβαση σε ψηφίδες μνήμης. Επιπλέον, το σύστημα διαθέτει γέφυρες προς ένα ή περισσότερους περιφερειακούς διαδρόμους, PCI ή PCI-X, όπου βρίσκονται συσκευές Ε/Ε· από τις σημαντικότερες είναι ο προσαρμογέας δικτύου και ο ελεγκτής μέσων αποθήκευσης, π.χ. ελεγκτής RAID.

Το κυριότερο γνώρισμα μιας τέτοιας σχεδίασης είναι ο *διαμοιρασμός* πόρων σε διάφορα επίπεδα: πυρήνες μοιράζονται επίπεδα της ιεραρχίας κρυφών μνημών, επεξεργαστές μοιράζονται εύρος ζώνης στον διάδρομο συστήματος, περιφερειακές συσκευές μοιράζονται εύρος ζώνης στον περιφερειακό διάδρομο και ανταγωνίζονται τους επεξεργαστές για πρόσβαση στη μνήμη. Ο διαμοιρασμός πόρων απλοποιεί τη σχεδίαση και διευκολύνει τον προγραμματισμό του συστήματος, ωστόσο μπορεί να έχει σημαντική επίπτωση στην επίδοση του, ανάλογα με το είδος των υπό εκτέλεση εφαρμογών. Εφαρμογές με καλή τοπικότητα αναφορών περνούν μεγάλο τμήμα του χρόνου εκτέλεσής τους χρησιμοποιώντας δεδομένα στην κρυφή μνήμη (σημείο **(a)**), ενώ εφαρμο-

γές με μεγάλες απαιτήσεις για δεδομένα (π.χ., [GKA+09]) επιβαρύνουν το μονοπάτι προς την κύρια μνήμη ((**b**)) και, καθώς ο όγκος των δεδομένων αυξάνεται, το μονοπάτι προς συσκευές αποθήκευσης. Σε συστοιχίες υπολογιστών, η πρόσβαση σε αποθηκευτικά μέσα παρέχεται συχνά μέσω ενός δικτύου αποθήκευσης (Storage Area Network) ή μέσω του δικτύου διασύνδεσης της συστοιχίας, όπως περιγράφεται λεπτομερέστερα στα επόμενα και στο § 0.2.1, οπότε η επίδοση εξαρτάται από τα μονοπάτια (**c**), (**d**) του Σχ. 1.

Αφενός η ανάγκη για εκτέλεση εφαρμογών απαιτητικών σε δεδομένα κι αφετέρου η συνεχής αύξηση της διαθέσιμης υπολογιστικής ισχύος ανά επεξεργαστή λόγω του αυξανόμενου αριθμού πυρήνων, αναδεικνύουν τη σημασία του συστήματος αποθήκευσης στη δυνατότητα κλιμάκωσης του συστήματος· η αποθηκευτική υποδομή πρέπει να αντεπεξέλθει στην ανάγκη των πυρήνων για δεδομένα [Gur09]. Χρειαζόμαστε μηχανισμούς για την αποδοτική *μετακίνηση* μεγάλου συνόλου δεδομένων ανάμεσα σε αποθηκευτικά μέσα και υπολογιστικούς πυρήνες, μέσω ενός δικτύου διασύνδεσης, με ελάχιστη επιβάρυνση στη λειτουργία του συστήματος.

Για την παροχή κλιμακούμενης αποθηκευτικής υποδομής σε συστοιχίες χρησιμοποιούνται συχνά συστήματα για μοιραζόμενη χρήση αποθηκευτικών συσκευών σε επίπεδο μπλοκ (block-level storage sharing). Ένα τέτοιο σύστημα επιτρέπει σε έναν αριθμό από κόμβους-πελάτες να έχουν πρόσβαση σε απομακρυσμένους δίσκους κόμβων-εξυπηρετητών, ως τοπικούς. Η αρχή λειτουργίας του παρουσιάζεται στο Σχ. 2: αιτήσεις Ε/Ε για μπλοκ μετασχηματίζονται σε δικτυακά μηνύματα, τα οποία περνούν στον εξυπηρετητή. Εφεξής, θα αναφερόμαστε σε τέτοια συστήματα και ως συστήματα δικτυακών συσκευών μπλοκ ή συστήματα nbd (network block devices).



Σχήμα 2: *Αρχή λειτουργίας συστήματος nbd*

Ιδανικά, ένα σύστημα nbd παρέχει ένα πολύ λεπτό στρώμα πρόσβασης σε απομακρυσμένες συσκευές, κλιμακούμενο με τον αριθμό τους, με ελάχιστη επιβάρυνση στους

πελάτες και τους εξυπηρετητές. Σε συστήματα SMP, η ανάγκη αυτή μεταφράζεται σε μειωμένο φόρτο CPU και μειωμένο ανταγωνισμό για μοιραζόμενους πόρους, δηλ. εύρος ζώνης στο διάδρομο κύριας μνήμης και τους περιφερειακούς διαδρόμους.

Ακόμη και σε ένα απομονωμένο σύστημα SMP, ο ανταγωνισμός για πρόσβαση στη μνήμη προκαλεί σημαντική επιβράδυνση της εκτέλεσης των εφαρμογών, καθώς διεργασίες σε διαφορετικούς επεξεργαστές συναγωνίζονται για πρόσβαση στο μονοπάτι (**b**) [Bel97, LVE00, ANP03]. Η πρόσβαση στη μνήμη από περιφερειακές συσκευές (διαδρομές (**c**), (**d**)) επιτείνει το πρόβλημα· σύγχρονα δίκτυα προσφέρουν ρυθμούς της τάξης των 10-40Gbps, αυξάνοντας την πίεση στο διάδρομο μνήμης και παρεμποδίζοντας τον υπολογισμό στους τοπικούς επεξεργαστές [Sch03], με εμφανή επιβάρυνση στην επίδοση [KK05, KK06]. Η σχεδίαση ενός συστήματος nbd οφείλει να στοχεύει στην ελαχιστοποίηση του κόστους απομακρυσμένης πρόσβασης σε δεδομένα, που αυξάνεται σημαντικά λόγω του ανταγωνισμού για υπολογιστικό χρόνο και εύρος ζώνης σε μοιραζόμενους διαδρόμους.

Τα σύγχρονα συστήματα nbd υστερούν από την άποψη αυτή. Συχνά βασίζονται στο TCP/IP, οπότε εισάγουν σημαντική επεξεργασία στην CPU του κόμβου. Επιπλέον, αντιμετωπίζουν την κύρια μνήμη ως κεντρικό πόρο: το μονοπάτι των δεδομένων διασχίζει πολλές φορές το διάδρομο μνήμης και τους περιφερειακούς διαδρόμους, ακόμη και όταν χρησιμοποιούνται προηγμένα δίκτυα διασύνδεσης με δυνατότητα Απομακρυσμένης Απευθείας Πρόσβασης στη Μνήμη [KKJ02, LPB04, LYP06]. Έτσι, επιβαρύνουν σημαντικά τους συμμετέχοντες κόμβους, ενώ η απόδοσή τους περιορίζεται λόγω κορεσμού των διαδρόμων Ε/Ε. Περισσότερα για την τρέχουσα κατάσταση στα § 0.2.1, § 0.6.

Η παρούσα διατριβή διερευνά τις επιπτώσεις του ανταγωνισμού για χρόνο CPU, εύρος ζώνης στο διάδρομο κύριας μνήμης και εύρος ζώνης στον περιφερειακό διάδρομο, σε κόμβους SMP που χρησιμοποιούνται ως αποθηκευτικοί εξυπηρετητές. Μελετά την κίνηση των δεδομένων σε ένα σύστημα nbd πάνω από δίκτυο Myrinet και αναδεικνύει την επίδρασή τους. Προτείνουμε τεχνικές για την κατασκευή αποδοτικών μονοπατιών δεδομένων ανάμεσα σε αποθηκευτικά μέσα και το δίκτυο, από την πλευρά του εξυπηρετητή, κι ανάμεσα στο δίκτυο και τους υπολογιστικούς πυρήνες, από την πλευρά του πελάτη. Επικεντρωνόμαστε σε βελτιστοποιήσεις λογισμικού συστήματος, με στόχο τον περιορισμό του ανταγωνισμού για πόρους και τη βελτίωση της ρυθμαπόδοσης του συστήματος.

Παρουσιάζουμε τη σχεδίαση και υλοποίηση του gmblock, ενός συστήματος διαμοιρασμού αποθηκευτικού χώρου που εκμεταλλεύεται τη διαθεσιμότητα επεξεργαστικής ισχύος και μονάδων μνήμης στον προσαρμογέα δικτύου για την κατασκευή απευθείας μονοπατιού δεδομένων ανάμεσα στο αποθηκευτικό μέσο και το δίκτυο. Η υλοποίησή του πάνω από Myrinet επιτρέπει την άμεση μετακίνηση δεδομένων από το δίσκο στο δίκτυο, χωρίς ενδιάμεσο αντίγραφο στη μνήμη του κόμβου και χωρίς την εμπλοκή του επεξεργαστή του. Η προσέγγιση αυτή μετριάζει τον ανταγωνισμό για μοιραζόμενους πόρους, βελτιώνει τη δυνατότητα κλιμάκωσης και επιτρέπει αύξηση του ρυθμού μεταφοράς έως και δύο φορές, σε σχέση με τις καθιερωμένες τεχνικές. Επιπλέον, ο αποθηκευτικός κόμβος δεν είναι απαραίτητο να χρησιμοποιείται αποκλειστικά για την εξυπηρέτηση αιτήσεων Ε/Ε· μπορεί να λειτουργεί επιπρόσθετα ως υπολογιστικός κόμβος, καθώς η απομακρυσμένη Ε/Ε δεν παρεμβαίνει στην εξέλιξη του τοπικού υπολογισμού. Η σχεδίαση του gmblock συνδυάζει υπάρχοντες μηχανισμούς αφαίρεσης που παρέχονται από το ΛΣ και το δίκτυο διασύνδεσης, επιτρέποντας την κατασκευή του μονοπατιού με γενικό τρόπο, χωρίς αλλαγές συγκεκριμένες για την υφιστάμενη αρχιτεκτονική. Έτσι είναι ανεξάρτητη του είδους της αποθηκευτικής συσκευής και διατηρεί τους μηχανισμούς απομόνωσης και προστασίας μνήμης του ΛΣ.

Η αρχική υλοποίηση ξεπερνά περιορισμούς εύρους ζώνης στο διάδρομο κύριας μνήμης και στον περιφερειακό διάδρομο, ωστόσο υπολείπεται των δυνατοτήτων του αποθηκευτικού μέσου και του δικτύου διασύνδεσης, λόγω του παράλληλου τρόπου μεταφοράς δεδομένων από αποθηκευτικά μέσα RAID και του περιορισμένου ποσού μνήμης που διαθέτει η κάρτα δικτύου. Για την καλύτερη προσαρμογή του gmblock στην υφιστάμενη αρχιτεκτονική και την υποστήριξη μεγαλύτερων αιτήσεων Ε/Ε χωρίς εμπλοκή του επεξεργαστή του κόμβου, προτείνουμε μια νέα κατηγορία λειτουργιών αποστολής πάνω από το Myrinet, που υποστηρίζουν *συγχρονισμό*· η σημασιολογία τους επιτρέπει στη δικτυακή μεταφορά να εξελίσσεται ελεγχόμενα, παράλληλα με τη μεταφορά δεδομένων από το δίσκο, επικαλύπτοντας τις δύο φάσεις για την ίδια αίτηση Ε/Ε. Η ενσωμάτωσή τους στο σύστημα βελτιώνει περαιτέρω το ρυθμό μεταφοράς που επιτυγχάνεται συγκριτικά με τη βασική έκδοση.

Στην πλευρά του πελάτη, βασιζόμαστε στην προγραμματισιμότητα του προσαρμογέα δικτύου και προτείνουμε τεχνικές μεταφοράς δεδομένων μπλοκ από το δίκτυο σε διάσπαρτες περιοχές φυσικής μνήμης χωρίς αντίγραφο και χωρίς εμπλοκή της CPU. Σε συνδυασμό με το προτεινόμενο μονοπάτι από την πλευρά του εξυπηρετητή, η σχεδία-

ση αυτή επιτρέπει τη μετακίνηση δεδομένων με μηδενικά αντίγραφα από άκρο σε άκρο. Η εγκατάσταση ενός παράλληλου συστήματος αρχείων, του Oracle OCFS2 πάνω από την υποδομή επιτρέπει την αξιολόγηση της επίδοσης με πραγματικά μετροπρογράμματα. Βρίσκουμε ότι η χρήση του άμεσου μονοπατιού γενικά ευνοεί την επίδοση, με την προϋπόθεση ότι το υποσύστημα αποθηκευτικών μονάδων παρέχει επαρκή ρυθμό μεταφοράς ώστε να μην γίνεται η στενωπός του συστήματος.

Η συνεισφορά της διατριβής αυτής συνοψίζεται στα εξής:

- Προτείνουμε απευθείας μονοπάτια Ε/Ε ανάμεσα σε αποθηκευτικές συσκευές και το δίκτυο, μειώνοντας την επίδραση του κορεσμού του διαδρόμου κύριας μνήμης και του περιφερειακού διαδρόμου σε αποθηκευτικούς εξυπηρετητές. Δείχνουμε πώς τέτοια μονοπάτια μπορούν να κατασκευαστούν με τρόπο ανεξάρτητο του αποθηκευτικού μέσου.

- Παρουσιάζουμε το gmblock, μια υλοποίηση του συστήματος πάνω από δίκτυο Myrinet. Αποτιμούμε πειραματικά την επίδοσή του, δείχνοντας ότι επιφέρει σημαντική αύξηση του ρυθμού απομακρυσμένης Ε/Ε κι επιτρέπει στον τοπικό υπολογισμό να συνεχίζει χωρίς σημαντική παρεμβολή.

- Ανακαλύπτουμε περιορισμούς σε διάφορα συστατικά μέρη του συστήματος, οι οποίοι μειώνουν την απόδοση των απευθείας μεταφορών. Δείχνουμε πώς η σχεδίαση μπορεί να ξεπεράσει ανάλογους περιορισμούς, χρησιμοποιώντας εναλλακτικά μονοπάτια με ενδιάμεση αποθήκευση σε απομονωτές στον περιφερειακό διάδρομο, συνεχίζοντας να παρακάμπτει το διάδρομο κύριας μνήμης.

- Προτείνουμε αρχιτεκτονικές αλλαγές ώστε η συγκεκριμένη σχεδίαση να εφαρμόζεται σε δίκτυα εκτός του Myrinet. Συζητούμε αλλαγές στη διαχείριση μνήμης από το ΛΣ, ώστε να υποστηρίζονται διακριτές περιοχές μνήμης, μερικές κοντύτερα στους επεξεργαστές, μερικές κοντύτερα στο δίκτυο.

- Προτείνουμε συγχρονισμένες λειτουργίες αποστολής ως επέκταση στη σημασιολογία του στρώματος ανταλλαγής μηνυμάτων του Myrinet, οι οποίες εκμεταλλεύονται την προγραμματισιμότητά του για να βελτιώσουν το χειρισμό μεγάλων αιτήσεων Ε/Ε, επιτρέποντας καλύτερη επικάλυψη των φάσεων Ε/Ε δικτύου και δίσκου.

- Παρουσιάζουμε βελτιστοποιήσεις στην πλευρά του πελάτη, με σκοπό την μεταφορά δεδομένων από το δίκτυο σε απομονωτές εφαρμογών χωρίς ενδιάμεσα αντίγραφα. Αυτή η σχεδίαση είναι, εξ όσων γνωρίζουμε, η πρώτη που επιτρέπει από άκρο σε άκρο μεταφορές μπλοκ με μηδενικά αντίγραφα, σε συνδυασμό με το προτεινόμενο μονοπάτι από την πλευρά του εξυπηρετητή.

- Μελετάμε τους συμβιβασμούς που γίνονται σχετικά με την δυνατότητα χρήσης κρυφής μνήμης (caching) και την προανάκτηση δεδομένων (prefetching) στην πλευρά του εξυπηρετητή, εγκαθιστώντας ένα παράλληλο σύστημα αρχείων πάνω από την υποδομή μας και αποτιμώντας την επίδοσή του με ποικίλα φορτία.

0.2 Υπόβαθρο

Το μέρος αυτό παρουσιάζει συνοπτικά το περιβάλλον υλικού και λογισμικού στο οποίο εστιάζει η παρούσα διατριβή. Αρχικά, συζητάμε τις βασικές αρχές των συστημάτων nbd και συχνά σενάρια χρήσης τους. Στη συνέχεια, εξετάζεται η λειτουργία των συστημάτων επικοινωνίας χώρου χρήστη, με έμφαση στο Myrinet, ώστε να οριστεί το πλαίσιο στο οποίο υλοποιούνται οι προτεινόμενες τεχνικές.

0.2.1 Συστήματα nbd και εφαρμογές

Η ανάγκη για μοιραζόμενη πρόσβαση επιπέδου μπλοκ σε κοινό αποθηκευτικό χώρο προκύπτει συχνά σε συστοιχίες υπολογιστών υψηλής επίδοσης. Μερικά από τα πιο συνηθισμένα σενάρια χρήσης είναι: *(a)* Η εγκατάσταση ενός παράλληλου συστήματος αρχείων μοιραζόμενου δίσκου, όπως το GPFS [SH02] και το GFS [PBB+99] *(b)* Η υποστήριξη παράλληλων εφαρμογών αρχιτεκτονικής μοιραζόμενου δίσκου, όπως είναι το σύστημα Oracle RAC και *(c)* περιβάλλοντα εικονικών μηχανών, όπου οι εικόνες των δίσκων των εικονικών μηχανών τοποθετούνται σε κοινό αποθηκευτικό χώρο, ώστε να είναι δυνατή η δυναμική μετακίνηση (live migration) των εικονικών μηχανών από περιέκτη σε περιέκτη.

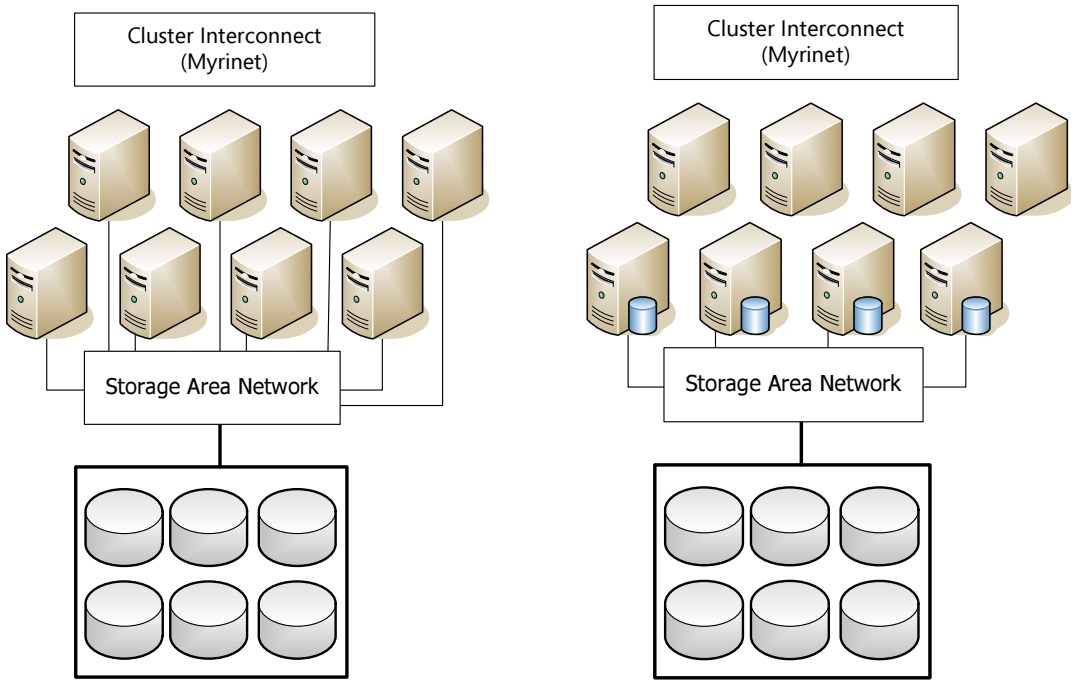
Παραδοσιακά, η απαίτηση για κοινό χώρο ικανοποιείται με χρήση εξειδικευμένου δικτύου αποθήκευσης (Storage Area Network - SAN) με βάση την αρχιτεκτονική Fibre

Channel (Σχ. 3(a)). Με τον τρόπο αυτό οι κοινοί δίσκοι εμφανίζονται ως τοπικά συνδεδεμένοι και χρησιμοποιούνται ανάλογα πρωτόκολλα, π.χ. SCSI για την πρόσβαση σε αυτούς.

Αυτή η προσέγγιση απαιτεί τη διατήρηση δύο δικτύων, ενός για πρόσβαση στα αποθηκευτικά μέσα κι ενός για τη διασύνδεση της συστοιχίας και την εκτέλεση παράλληλων εφαρμογών. Το SAN χρειάζεται να κλιμακωθεί σε μεγάλο αριθμό κόμβων, ενώ αυξάνεται το κόστος ανά κόμβο. Επιπλέον, το εύρος ζώνης προς τους δίσκους παραμένει σταθερό και δεν αυξάνεται καθώς προστίθενται νέοι κόμβοι στο σύστημα. Για να μειωθεί το κόστος ακολουθείται συχνά μια υβριδική προσέγγιση: ένας αριθμός κόμβων παραμένει φυσικά συνδεδεμένος στο SAN (κόμβοι αποθήκευσης) και *εξάγει* τους δίσκους για πρόσβαση επιπέδου μπλοκ μέσω του δικτύου διασύνδεσης στους υπόλοιπους (Σχ. 3(b)). Συνεχίζοντας στην ίδια λογική, το SAN μπορεί να εξαλειφθεί εντελώς, οπότε το σύνολο των κόμβων συνεισφέρει μέρος τοπικά συνδεδεμένων αποθηκευτικών μέσων για τη δημιουργία ενός *εικονικού*, μοιραζόμενου χώρου αποθήκευσης (Σχ. 3(c)).

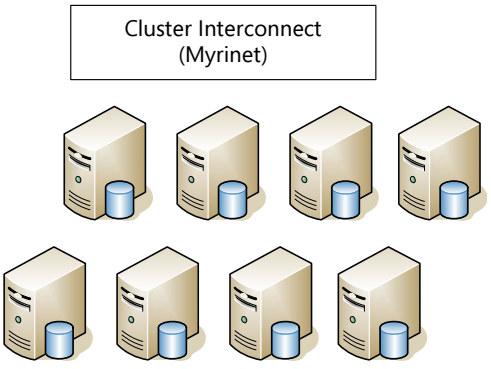
Ο ακρογωνιαίος λίθος μιας τέτοιας προσέγγισης είναι ένα σύστημα δικτυακής συσκευής μπλοκ (network block device - nbd) (Σχ. 4) το οποίο επιτρέπει πρόσβαση σε απομακρυσμένες αποθηκευτικές συσκευές μέσω του δικτύου διασύνδεσης. Τέτοια συστήματα περιλαμβάνουν το στρώμα NSD (Network Shared Disks), μέρος του GPFS, το NBD (Network Block Device), μέρος του πυρήνα του Linux και το GNBD, μια βελτιωμένη έκδοσή του για το GFS της Red Hat.

Οι υλοποιήσεις αυτές βασίζονται στο TCP/IP, οπότε εμπλέκουν στοίβα πρωτοκόλλων που εκτελείται στον επεξεργαστή του συστήματος και δεν εκμεταλλεύονται προηγμένα χαρακτηριστικά των δικτύων διασύνδεσης, όπως τη δυνατότητα για ανταλλαγή μηνυμάτων χωρίς αντίγραφα και απομακρυσμένη απευθείας πρόσβασης στη μνήμη (RDMA). Από την άλλη πλευρά, υπάρχουν ερευνητικές προσπάθειες για την εκμετάλλευση αυτών των χαρακτηριστικών, που αναλύονται διεξοδικότερα στο § 0.6. Ωστόσο, το πρόβλημα παραμένει ότι παρόλο που απλοποιείται σημαντικά η απαιτούμενη στοίβα πρωτοκόλλων, η κίνηση των δεδομένων από τα αποθηκευτικά μέσα στους υπολογιστικούς πυρήνες ακολουθεί μονοπάτια που αυξάνουν τον ανταγωνισμό για μοιραζόμενους αρχιτεκτονικούς πόρους, εισάγοντας σημαντική επιβάρυνση.



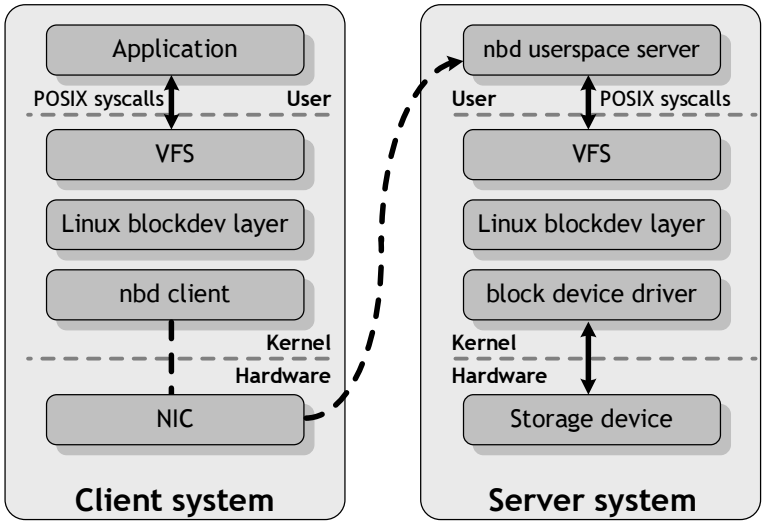
(a) Όλοι οι κόμβοι συνδεδεμένοι στο SAN

(b) Κόμβοι αποθήκευσης στο SAN



(c) Μόνο τοπικές αποθηκευτικές μονάδες

Σχήμα 3: *Διασύνδεση κόμβων συστοιχίας με συσκευές αποθήκευσης*



Σχήμα 4: *Γενικό σύστημα nbd*

0.2.2 Συστήματα επικοινωνίας χώρου χρήστη

Βασικές αρχές

Η πρόοδος στον τομέα των δικτύων υψηλής επίδοσης έχει επιφέρει σημαντική αύξηση στον προσφερόμενο ρυθμό μεταφοράς δεδομένων, που βρίσκεται στην περιοχή των 2-40Gbps και μείωση στο χρόνο αρχικής απόκρισης, ο οποίος κυμαίνεται από 0.3–1.0μs, ανάλογα με το μέγεθος του δικτύου.

Ωστόσο, η παροχή ανάλογων επιδόσεων στις τελικές εφαρμογές δυσχεραίνεται από την ύπαρξη πολύπλοκης στοίβας πρωτοκόλλων, όπως το TCP/IP στον πυρήνα του ΛΣ. Για να μειώσουν την επιβάρυνση που προκαλεί η εμπλοκή του ΛΣ στο κρίσιμο μονοπάτι, πολλά σύγχρονα δίκτυα, όπως το SCI [Hel99], το Quadrics [PcFH$^+$01], το Infiniband [Inf00] και το Myrinet [BCF$^+$95] ακολουθούν αρχιτεκτονική επικοινωνίας χώρου χρήστη (user level networking). Το μοντέλο αυτό επιτρέπει τον άμεσο έλεγχο του προσαρμογέα δικτύου από διεργασίες χώρου χρήστη.

Εφόσον το ΛΣ παρακάμπτεται, ο ρόλος του πυρήνα αναλαμβάνεται από ένα συνδυασμό βιβλιοθήκης χώρου χρήστη, και υλικολογισμικού (firmware) που εκτελείται πάνω στην κάρτα δικτύου· η επικοινωνία ανάμεσά τους εγκαθίσταται από προνομιούχο κώδικα που εκτελείται ως δομικό στοιχείο (module) του ΛΣ (Σχ. 5).

Η εφαρμογή αποκτά έλεγχο της δικτυακής διεπαφής μέσω απεικονίσεων στο χώρο εικονικής μνήμης της. Εφόσον το ΛΣ και ο επεξεργαστής του κόμβου αφαιρούνται από

το κρίσιμο μονοπάτι, η λειτουργικότητά τους αναλαμβάνεται από την ίδια την κάρτα, η οποία χρειάζεται να είναι προγραμματιζόμενη σε κάποιο βαθμό, αναλόγως με το πρωτόκολλο επικοινωνίας χώρου χρήστη που χρησιμοποιείται.

Κατά τη σχεδίαση ενός τέτοιου πρωτοκόλλου προκύπτουν ζητήματα προγραμματιστικής σημασιολογίας, μετακίνησης δεδομένων, μετάφρασης διευθύνσεων και προστασίας μνήμης. Στα επόμενα, παρουσιάζεται εν συντομία πώς αυτά τα θέματα αντιμετωπίζονται στη σχεδίαση του στρώματος επικοινωνίας GM για δίκτυα Myrinet, που αποτελεί και το περιβάλλον υλοποίησης των προτεινόμενων τεχνικών αυτής της διατριβής.

Υλοποίηση επικοινωνίας χώρου χρήστη στο Myrinet/GM

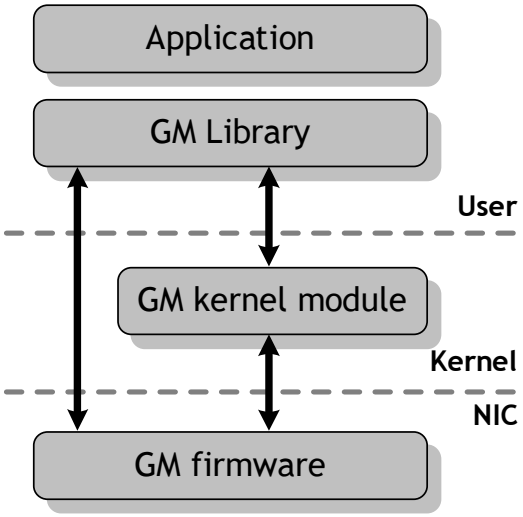
Το Myrinet είναι μια δικτυακή υποδομή υψηλού ρυθμού μεταφοράς και χαμηλού χρόνου αρχικής απόκρισης για συστοιχίες υπολογιστών, που χρησιμοποιεί τεχνικές επικοινωνίας χώρου χρήστη [BRB98b] για να αφαιρέσει το ΛΣ από το κρίσιμο μονοπάτι της επικοινωνίας. Η στοίβα λογισμικού του Myrinet όταν χρησιμοποιείται το στρώμα επικοινωνίας GM [Myr03] παρουσιάζεται στο Σχ. 5.

Η εφαρμογή αποκτά τον έλεγχο του προσαρμογέα δικτύου μέσω απεικονίσεων της μνήμης του προσαρμογέα στο δικό της χώρο εικονικών διευθύνσεων· όταν ολοκληρωθεί η εγκατάσταση των απεικονίσεων, χρησιμοποιεί μη προνομιούχες εντολές πρόσβασης στη μνήμη για να εκτελέσει λειτουργίες δικτυακής Ε/Ε.

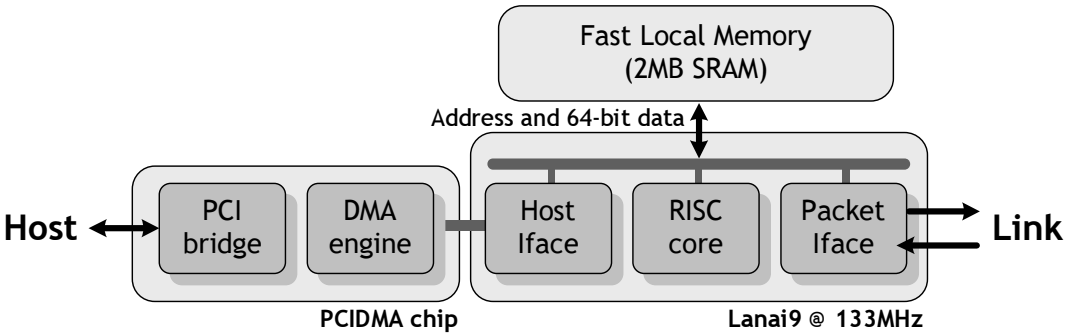
Ο προσαρμογέας δικτύου του Myrinet συνδέεται στον περιφερειακό διάδρομο, στην υποδομή που εξετάζουμε τύπου PCI/PCI-X. Διαθέτει έναν επεξεργαστή τύπου RISC, που ονομάζεται Lanai και ένα μικρό ποσό (2MB) μνήμης SRAM που χρησιμοποιείται από το Lanai και τρεις διαφορετικές μηχανές DMA: μία για κίνηση δεδομένων από τη μνήμη του κόμβου στη μνήμη του Lanai και δύο για ανταλλαγή δεδομένων ανάμεσα στη μνήμη του Lanai και το φυσικό μέσο.

Το Σχ. 6 παρουσιάζει τα κύρια στοιχεία μιας κάρτας Myrinet τύπου M3F-PCI64B-2, η οποία χρησιμοποιεί χωριστές ψηφίδες για κάθε μονάδα. Στη συγκεκριμένη εργασία χρησιμοποιούνται κάρτες M3F2-PCIXE-2, όπου όλη η λειτουργικότητα ενσωματώνεται σε μία ψηφίδα LanaiX, η οποία υποστηρίζει δύο συνδέσμους πακέτων.

Για να είναι εφικτή η επικοινωνία χωρίς αντίγραφα, το GM υποστηρίζει την απεικόνιση από τις διεργασίες μερών της μνήμης του Lanai, που ονομάζονται *θύρες* του GM.



Σχήμα 5: *Στοίβα Myrinet/GM*

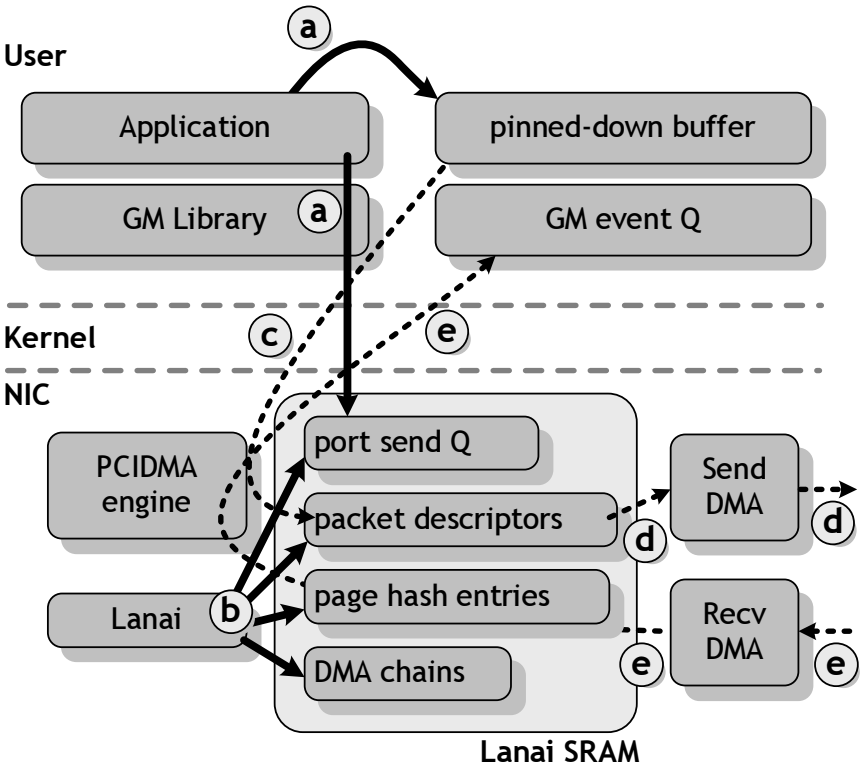


Σχήμα 6: *Κυριότερα μέρη προσαρμογέα δικτύου Myrinet*

Κάθε θύρα αποτελεί άκρο επικοινωνίας για την εφαρμογή. Αποτελείται από ένα μη προνομιούχο τμήμα, το οποίο περιέχει ουρές αποστολής και λήψης που τροποποιούνται απευθείας από τη διεργασία κι ένα προνομιούχο τμήμα, που μεταβάλλεται από το υλικολογισμικό και τον οδηγό χώρου πυρήνα.

Το υλικολογισμικό του GM ελέγχει περιοδικά τις ουρές, εντοπίζει νέες αιτήσεις των εφαρμογών και χρησιμοποιεί τις μηχανές DMA για να τις εξυπηρετήσει. Το GM παρέχει αξιόπιστη επικοινωνία σημείου-προς-σημείο χωρίς σύνδεση ανάμεσα σε διαφορετικές θύρες, πολυπλέκοντας δεδομένα διαφορετικών θυρών πάνω από συνδέσεις ανάμεσα σε ζεύγη κόμβων. Ένα πρωτόκολλο δικτύου «Go back N» χρησιμοποιείται για την υλοποίηση αξιόπιστων συνδέσεων.

Η λειτουργικότητα του υλικολογισμικού οργανώνεται σε τέσσερις μηχανές καταστάσεων, τις SDMA, SEND, RECV και RDMA, κάθε μία από τις οποίες αναλαμβάνει μέρος της επεξεργασίας του δικτυακού πρωτοκόλλου. Η μηχανή SDMA ελέγχει τις ανοιχτές



Σχήμα 7: *Αποστολή μηνύματος στο Myrinet/GM*

θύρες, εντοπίζει νέες αιτήσεις αποστολής/λήψης και προγραμματίζει τη μηχανή DMA για μεταφορά από τη μνήμη του κόμβου στη μνήμη του Lanai. Η μηχανή SEND δέχεται πακέτα που παράγει η μηχανή SDMA και προγραμματίζει λειτουργίες DMA για προώθηση στο φυσικό μέσο. Επιπλέον, προωθεί πακέτα θετικής ή αρνητικής επιβεβαίωσης (ACK/NACK) του πρωτοκόλλου. Η μηχανή RECV χειρίζεται εισερχόμενα πακέτα από το δίκτυο και παράγει ανάλογα πακέτα επιβεβαίωσης. Τέλος, η μηχανή RDMA δέχεται εισερχόμενα πακέτα από τη μηχανή RECV, τα ταιριάζει με απομονωτές εισερχόμενων μηνυμάτων της διεργασίας χώρου χρήστη και εκκινεί δοσοληψίες DMA για τη μετακίνησή τους στις τελικές τους θέσεις, στην κύρια μνήμη του συστήματος.

Στα επόμενα, παρουσιάζονται περιληπτικά τα βασικά βήματα για την ολοκλήρωση μιας διαδικασίας αποστολής μηνύματος μέσω του GM (Σχ. 7), κάνοντας αναφορά στους βασικούς μηχανισμούς ελέγχου, κίνησης δεδομένων, μετάφρασης διευθύνσεων και προστασίας μνήμης που αυτό παρέχει.

Οι συνεχείς γραμμές αφορούν Προγραμματιζόμενη Ε/Ε (Programmable I/O - PIO), η οποία εμπλέκει είτε τη CPU του κόμβου είτε το Lanai, ενώ οι διακεκομμένες αναφέρο-

νται σε δοσοληψίες DMA.

Η βασική κλήση αποστολής του GM `gm_send_with_callback()` συνεπάγεται τις εξής φάσεις εξυπηρέτησης, οι οποίες αντιστοιχούν στα βήματα **(a)**-**(e)** του Σχ. 7.

(a) Δέσμευση του απομονωτή, δημιουργία αίτησης αποστολής Η εφαρμογή υπολογίζει το μήνυμα προς αποστολή σε καθηλωμένο (pinned down) απομονωτή χώρου χρήστη. Αυτό εξασφαλίζεται με κατάλληλη κλήση προς τον οδηγό του GM, ώστε οι αντίστοιχες σελίδες να μην μετακινούνται ποτέ από την κύρια μνήμη. Στη συνέχεια δημιουργεί κατάλληλο αίτημα αποστολής, που περιέχει την εικονική διεύθυνση του απομονωτή, σε ανάλογη ουρά στη μνήμη του Lanai.

(b) Μετάφραση διευθύνσεων και αρχικοποίηση DMA Η μηχανή SDMA εντοπίζει τη νέα αίτηση και προγραμματίζει τη μηχανή DMA για μεταφορά των δεδομένων από την κύρια μνήμη στη μνήμη του Lanai. Η μεταφορά χρειάζεται τη φυσική διεύθυνση του απομονωτή. Για τη διατήρηση του μηχανισμού προστασίας μνήμης του ΛΣ, το GM τηρεί πίνακες σελίδων για τη μετάφραση εικονικών διευθύνσεων σε φυσικές, η οποία αναλαμβάνεται εξ ολοκλήρου από τον προσαρμογέα δικτύου. Οι πίνακες βρίσκονται στην κύρια μνήμη. Η κάρτα δικτύου κρατά μέρος τους αποθηκευμένο στη μνήμη του Lanai (page hash entries), ώστε να μειώσει το κόστος πρόσβασης σε αυτούς.

(c) Δοσοληψία DMA Τα δεδομένα του μηνύματος έρχονται στη μνήμη του Lanai, μέσα σε κατάλληλους περιγραφητές μηνυμάτων. Όταν η διαδικασία ολοκληρωθεί, ενημερώνεται η μηχανή SEND.

(d) Εισαγωγή πακέτων στο δίκτυο Η μηχανή DMA της διεπαφής πακέτων προγραμματίζεται να εισάγει δεδομένα από τη μνήμη του Lanai στο φυσικό σύνδεσμο.

(e) Επιβεβαίωση από την απομακρυσμένη πλευρά Λαμβάνονται πακέτα επιβεβαίωσης (ACK) από την απομακρυσμένη πλευρά, έως ότου το σύνολο των πακέτων που αφορούν το μήνυμα έχει παραδοθεί. Τότε, ενημερώνεται η διεργασία μέσω ουράς συμβάντων που τηρείται στην κύρια μνήμη του κόμβου. Η διεργασία μπορεί να ελέγχει περιοδικά την ουρά αυτή (polling) ή να ζητήσει να κοιμηθεί μέσα στον πυρήνα. Στην περίπτωση αυτή, η κάρτα δικτύου ενημερώνει με διακοπή υλικού τον επεξεργαστή για την ολοκλήρωση της διαδικασίας αποστολής.

Πρέπει να τονιστεί ότι η αποστολή και λήψη μηνυμάτων με το GM είναι διαδικασία δύο βημάτων:

Δοσοληψία DMA από τη μνήμη του κόμβου στο Lanai: γίνεται μετάφραση διευθύνσεων, ενεργοποιείται η μηχανή DMA για τον περιφερειακό διάδρομο και τα δεδομένα μεταφέρονται από την κύρια μνήμη στην SRAM του Lanai

Δοσοληψία DMA από το Lanai στο καλώδιο: Δεδομένα πακέτων ανακτώνται από τη μνήμη του Lanai και εισάγονται στο δίκτυο με προορισμό τον απομακρυσμένο κόμβο.

0.3 Σχεδίαση και υλοποίηση του gmblock

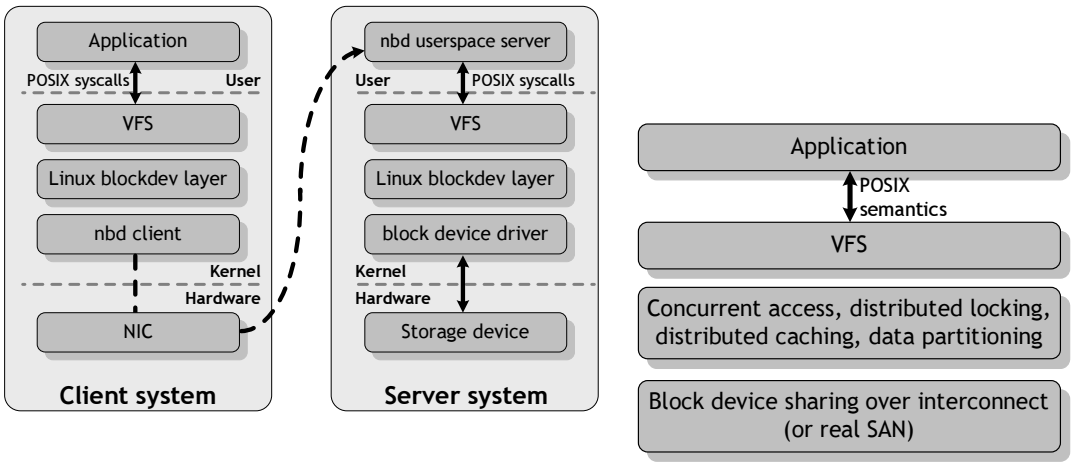
Το μέρος αυτό παρουσιάζει τη σχεδίαση και υλοποίηση του gmblock. Συζητάμε την επιβάρυνση που εισάγουν καθιερωμένες προσεγγίσεις βασισμένες στο TCP/IP ή σε τεχνικές RDMA και πώς η σχεδίαση του gmblock εξελίσσεται από αυτές. Στη συνέχεια παρουσιάζονται οι αλλαγές που απαιτούνται στο GM και τον πυρήνα του Linux για να υποστηρίξουν μια πρότυπη υλοποίηση, η οποία αποτιμάται πειραματικά. Τέλος, συζητούνται θέματα που αφορούν τη σχεδίαση του gmblock, όπως ο διαμοιρασμός δομημένων δεδομένων, ο συνδυασμός των προτεινόμενων τεχνικών με δίκτυα εκτός Myrinet και η χρήση τους για σχεδιάσεις χαμηλής κατανάλωσης ενέργειας.

0.3.1 Σχεδίαση του μηχανισμού nbd του gmblock

Παραδοσιακές σχεδιάσεις nbd

Η βασική αρχή πίσω από μια υλοποίηση nbd πελάτη-εξυπηρετητή παρουσιάζεται στο Σχ. 8(a). Ο πελάτης βρίσκεται συνήθως στο χώρο πυρήνα και εξάγει μια εικονική συσκευή στο υπόλοιπο σύστημα. Οι αιτήσεις ενθυλακώνονται σε δικτυακά μηνύματα και προωθούνται στον εξυπηρετητή, ο οποίος εκτελείται στο χώρο χρήστη και χρησιμοποιεί καθιερωμένες κλήσεις συστήματος για Ε/Ε με την τοπική συσκευή μπλοκ.

Ο ψευδοκώδικας ενός γενικού εξυπηρετητή φαίνεται στο Σχ. 9. Εμπλέκονται τέσσερα βασικά βήματα στην εξυπηρέτηση μιας αίτησης: **(a)** Λήψη μηνύματος που περιέχει την



(a) Γενικό σύστημα nbd

(b) Παράλληλο σύστημα αρχείων πάνω από nbd

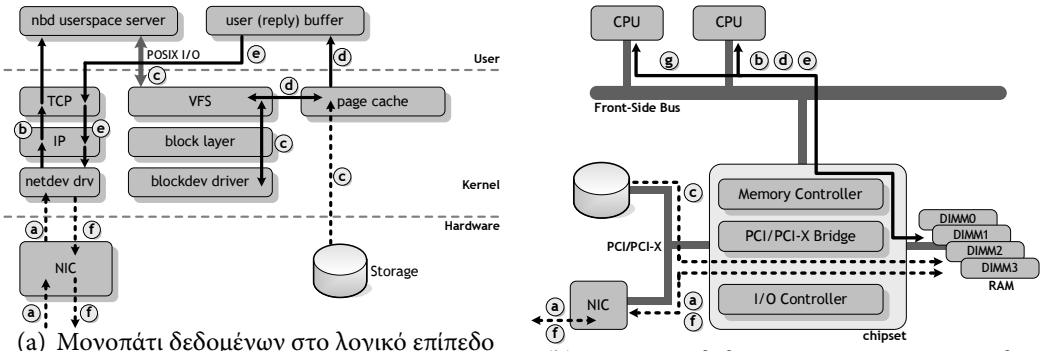
Σχήμα 8: *Εκτέλεση παράλληλου συστήματος αρχείων πάνω από υποδομή nbd*

αίτηση (**b**) Εντοπισμός των αντίστοιχων μπλοκ, π.χ. με `lseek()` (**c**) Μεταφορά των δεδομένων σε απομονωτή χώρου χρήστη (**d**) αποστολή στον πελάτη.

```
initialize_interconnect();
fd = open_block_device();
reply = allocate_memory_buffer();
for (;;) {
      cmd = recv_cmd_from_interconnect();
      lseek(fd, cmd->start, SEEK_SET);
      switch (cmd->type) {
         case READ_BLOCK:
            read(fd, &reply->payload, cmd->len);
         case WRITE_BLOCK:
            write(fd, &req->payload, cmd->len);
      }
      insert_packet_headers(&reply, cmd);
      send_over_net(reply, reply->len);
}
```

Σχήμα 9: *Ψευδοκώδικας εξυπηρετητή nbd*

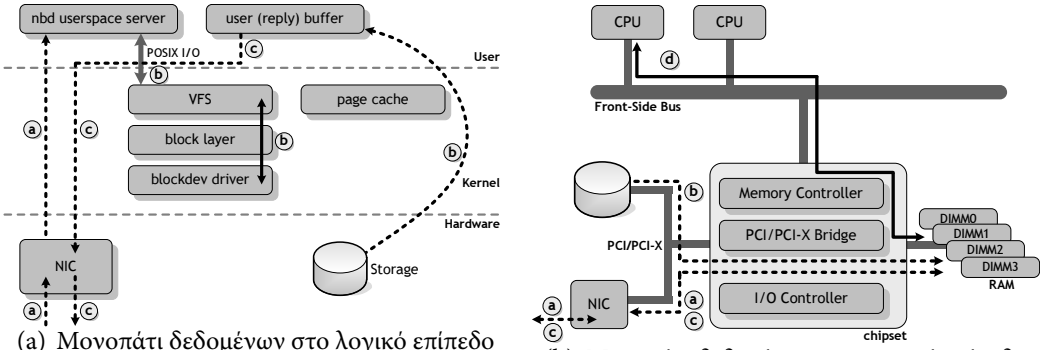
Η επιβάρυνση των λειτουργιών εξαρτάται από το είδος του δικτύου διασύνδεσης και την προσφερόμενη λειτουργικότητα. Για έναν εξυπηρετητή βασισμένο στο TCP/IP, η μεταφορά σε λογικό επίπεδο παρουσιάζεται στο Σχ. 10(a). Οι συνεχείς γραμμές υποδηλώνουν προγραμματιζόμενη Ε/Ε, οι διακεκομμένες Ε/Ε με DMA: (**a**) Εισερχόμενα πλαίσια τοποθετούνται με DMA στη μνήμη (**b**) Εκτελείται η στοίβα TCP/IP, το μήνυμα



(a) Μονοπάτι δεδομένων στο λογικό επίπεδο

(b) Μονοπάτι δεδομένων στο φυσικό επίπεδο

Σχήμα 10: *Εξυπηρετητής nbd βασισμένος σε TCP/IP*



(a) Μονοπάτι δεδομένων στο λογικό επίπεδο

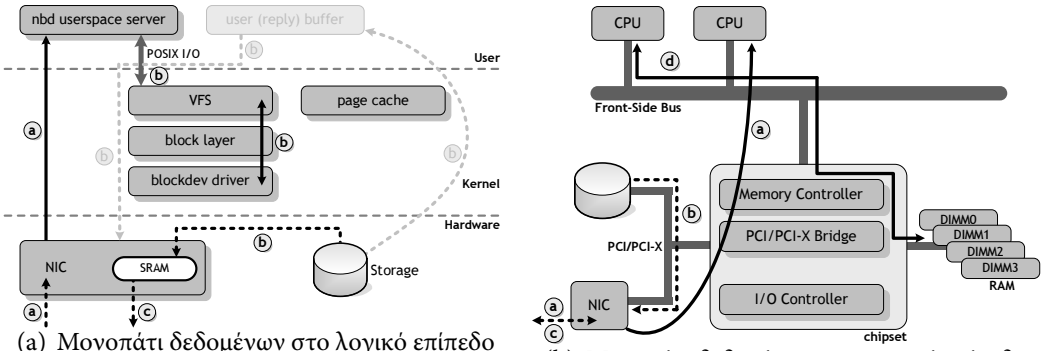
(b) Μονοπάτι δεδομένων στο φυσικό επίπεδο

Σχήμα 11: *Εξυπηρετητής nbd βασισμένος στο GM*

συντίθεται και αντιγράφεται σε απομονωτή χώρου χρήστη, ενεργοποιείται η διεργασία του εξυπηρετητή nbd. Ο εξυπηρετητής nbd εκτελεί μια κλήση συστήματος `read()` για τα δεδομένα, στη γενική περίπτωση αυτή χρησιμοποιεί την κρυφή μνήμη σελίδων (page cache) του πυρήνα (**c**) ο οδηγός συσκευής μεταφέρει τα δεδομένα μπλοκ με DMA στην κρυφή μνήμη σελίδων του πυρήνα (**d**) τα δεδομένα αντιγράφονται από εκεί στον απομονωτή χώρου χρήστη, ώστε τελικά (**e**) ο εξυπηρετητής να ζητήσει την αποστολή τους, οπότε αντιγράφονται πίσω στη μνήμη του πυρήνα, εκτελείται η στοίβα TCP/IP, σχηματίζονται πλαίσια Ethernet, τα οποία (**f**) η κάρτα δικτύου παραλαμβάνει με DMA και τοποθετεί στο καλώδιο.

Στο Σχ. 10(b), παρουσιάζεται η ίδια διαδικασία σε φυσικό επίπεδο· υπάρχει αντιστοιχία ένα-προς-ένα στα βήματα των σχημάτων 10(a), 10(b).

Το μονοπάτι δεδομένων που περιγράψαμε περιέχει πολλαπλά βήματα αντιγραφής των δεδομένων. Ο αριθμός τους θα μπορούσε να μειωθεί με διάφορες τεχνικές, π.χ. E/E με απεικόνιση μνήμης (`mmap()`).



(a) Μονοπάτι δεδομένων στο λογικό επίπεδο

(b) Μονοπάτι δεδομένων στο φυσικό επίπεδο

Σχήμα 12: *Προτεινόμενος εξυπηρετητής gmblock*

Εναλλακτικά, θα μπορούσε να εξαλειφθεί ένα αντίγραφο παρακάμπτοντας την κρυφή μνήμη του πυρήνα: ο εξυπηρετητής nbd χρησιμοποιεί τη λειτουργικότητα `O_DIRECT` του προτύπου POSIX για να ζητήσει μεταφορά των δεδομένων απευθείας στους απομονωτές χώρου χρήστη. Το στρώμα συσκευών μπλοκ του Linux παρέχει μια γενική υλοποίησή άμεσης E/E, η οποία προσδιορίζει τις φυσικές διευθύνσεις των απομονωτών και εκτελεί DMA σε αυτές.

Στην παραπάνω σχεδίαση, ακόμη και με ένα αντίγραφο στον πυρήνα, τα δεδομένα διασχίζουν δύο φορές τον περιφερειακό διάδρομο και τέσσερις το διάδρομο κύριας μνήμης, ενώ εμπλέκεται κι η CPU στη μετακίνησή τους. Το πρόβλημα μετριάζεται αν χρησιμοποιηθεί δίκτυο με δυνατότητα επικοινωνίας χώρου χρήστη, όπως το Myrinet. Στην περίπτωση αυτή (Σχ. 11(a), 11(b)) μπορούν να εξαλειφθούν τα περισσότερα αντίγραφα, ωστόσο και πάλι τα δεδομένα διασχίζουν δύο φορές τον διάδρομο κύριας μνήμης και τον περιφερειακό διάδρομο, καταναλώνοντας εύρος ζώνης και επηρεάζοντας την εξέλιξη του τοπικού υπολογισμού.

GMBlock: Εναλλακτικό μονοπάτι με παράκαμψη κύριας μνήμης

Προτείνουμε ένα συντομότερο μονοπάτι δεδομένων, που παρακάμπτει εντελώς την κύρια μνήμη. Για την εξυπηρέτηση μιας αίτησης E/E, χρειάζεται μόνο να μετακινηθούν δεδομένα από το αποθηκευτικό μέσο στο δίκτυο. Η κίνηση των δεδομένων στο φυσικό επίπεδο παρουσιάζεται στο Σχ. 12(b): **(a)** παραλαβή αίτησης από την κάρτα δικτύου **(b)** επεξεργασία της αίτησης από τον εξυπηρετητή, με άμεση μεταφορά δεδομένων από το μέσο στην κάρτα δικτύου του Myrinet **(c)** αποστολή δεδομένων στον απομακρυσμένο κόμβο.

Η υλοποίηση του μονοπατιού αυτού λύνει τα περισσότερα από τα προβλήματα που αναφέρθηκαν στα προηγούμενα: εκμεταλλεύεται πλήρως το εύρος ζώνης του περιφερειακού διαδρόμου, εφόσον το μονοπάτι τον διασχίζει μία φορά, τα δεδομένα δεν αποθηκεύονται ενδιάμεσα στη RAM του κόμβου, δεν καταναλώνεται εύρος ζώνης μνήμης λόγω απομακρυσμένης Ε/Ε και εξαλείφεται ο ανταγωνισμός για πρόσβαση στη μνήμη.

Επιπρόσθετα, αυτή η σχεδίαση αναγνωρίζει ότι το μονοπάτι για απομακρυσμένη Ε/Ε μπορεί να είναι χωριστό από την κύρια μνήμη· η ανάγκη για απομονωτές στην κύρια μνήμη προκύπτει από την προγραμματιστική σημασιολογία του GM και του πυρήνα, όχι από εγγενείς ιδιότητες της αρχιτεκτονικής. Για την υποστήριξη του προτεινόμενου μονοπατιού, απαιτούνται επεκτάσεις στο GM και τον πυρήνα του ΛΣ έτσι ώστε να είναι δυνατή η απευθείας μεταφορά ανάμεσα στο μέσο αποθήκευσης και το δίκτυο, χρησιμοποιώντας τις ήδη διαθέσιμες αφαιρετικές δομές της επικοινωνίας χώρου χρήστη και του μηχανισμού κλήσεων συστήματος, ώστε να ελαχιστοποιηθούν οι απαιτούμενες αλλαγές στον πηγαίο κώδικα του εξυπηρετητή nbd.

Ξεκινάμε από τον κώδικα του Σχ. 9. Στην περίπτωση του GM ο απομονωτής δεσμεύεται στην κύρια μνήμη ώστε να μπορεί να χρησιμοποιηθεί απευθείας από την κάρτα δικτύου (`gm_dma_malloc()`), η μεταβλητή `reply` περιέχει την εικονική διεύθυνσή του. Ωστόσο, αν αντί να δεσμευτεί στην κύρια μνήμη, είχε δεσμευτεί επάνω στη μνήμη SRAM του προσαρμογέα, η επιθυμητή σημασιολογία θα μπορούσε να εκφραστεί με την κλήση `read()` που ακολουθεί, χωρίς αλλαγή: θα συνέχιζε να σημαίνει «διάβασε τα δεδομένα μπλοκ και αποθήκευσέ τα στον απομονωτή με εικονική διεύθυνση `reply`», ωστόσο τώρα το μονοπάτι δεδομένων θα ήταν διαφορετικό.

Για να εξασφαλιστεί ότι τα δεδομένα δεν θα περάσουν από την κρυφή μνήμη σελίδων του πυρήνα, η κλήση `read()` πρέπει να συνδυαστεί με *άμεση* Ε/Ε, χρησιμοποιώντας το μηχανισμό `O_DIRECT` του POSIX. Στην περίπτωση αυτή, ο πυρήνας παρακάμπτει την κρυφή μνήμη, το υποσύστημα άμεσης Ε/Ε μετατρέπει την εικονική διεύθυνση σε φυσική διεύθυνση η οποία ανήκει στο χώρο διευθύνσεων PCI του προσαρμογέα Myrinet, οπότε η αίτηση προς τον οδηγό συσκευής μπλοκ αφορά στην SRAM του προσαρμογέα. Η δοσοληψία DMA που γίνεται στη συνέχεια από το αποθηκευτικό μέσο, προκαλεί την μετακίνηση των δεδομένων απευθείας στην κάρτα. Για την αποστολή των δεδομένων μέσω GM, παραλείπεται το πρώτο μισό, το τμήμα της μεταφοράς των δεδομένων από τη RAM στην SRAM του προσαρμογέα και πραγματοποιείται μόνο το βήμα της μετα-

φοράς από την SRAM στο φυσικό μέσο.

Για την υλοποίηση του μονοπατιού δεν απαιτούνται αλλαγές στον κώδικα του εξυπηρετητή. Συνεχίζει να χρησιμοποιεί κλήσεις `read()`/`write()` και `gm_send()`, ενώ η επιθυμητή σημασιολογία προκύπτει από τον τρόπο που συνδυάζονται το στρώμα συσκευών μπλοκ του Linux, το υποσύστημα εικονικής μνήμης και η δυνατότητα επικοινωνίας χώρου χρήστη του GM, για να κατασκευαστεί το μονοπάτι δεδομένων.

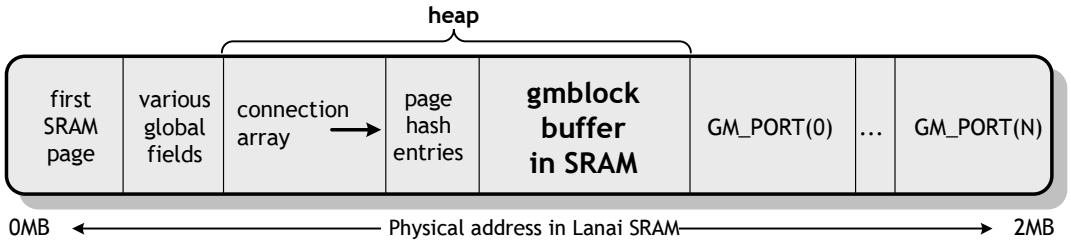
0.3.2 Λεπτομέρειες υλοποίησης

Η υλοποίηση του προτεινόμενου μονοπατιού εμπλέκει δύο διακριτά υποσυστήματα: πρώτον, επεκτείνουμε το GM ώστε να υποστηρίζει απομονωτές μηνυμάτων μέσα στην SRAM του Lanai, δεύτερον επεκτείνουμε το μηχανισμό εικονικής μνήμης του πυρήνα ώστε να υποστηρίζει απευθείας E/E με περιοχές μνήμης στο χώρο διευθύνσεων του PCI.

Υποστήριξη GM για απομονωτές στην SRAM

Επεκτείνουμε το GM ώστε να επιτρέπει τη δέσμευση, την απεικόνιση και το χειρισμό απομονωτών στην SRAM από εφαρμογές χώρου χρήστη, διατηρώντας ταυτόχρονα το μηχανισμό απομόνωσης και προστασίας μνήμης του GM και του πυρήνα του Linux.

Για τη δέσμευση των απομονωτών επεκτείνουμε το υλικολογισμικό του GM ώστε να δεσμεύει μια ενιαία περιοχή της μνήμης κατά την αρχικοποίησή του (Σχ. 13). Στην πειραματική μας πλατφόρμα, μπορέσαμε να δεσμεύσουμε μέχρι 700KB από τα 2MB της κάρτας για χρήση του gmblock.



Σχήμα 13: *Χάρτης μνήμης του Lanai με δέσμευση του απομονωτή του gmblock*

Για την απεικόνιση στο χώρο μνήμης των διεργασιών επεκτείνονται η βιβλιοθήκη χώρου χρήστη κι ο οδηγός συσκευής του GM, ώστε να εγκαθίσταται η απαιτούμενη απει-

κόνιση αφού επαληθευτεί η ορθότητα της αίτησης.

Τέλος, για να είναι δυνατή η συμμετοχή του απομονωτή σε λειτουργίες αποστολής/λήψης μηνυμάτων μεταβάλλουμε τη βιβλιοθήκη χώρου χρήστη και το υλικολογισμικό του GM. Η βιβλιοθήκη εντοπίζει πότε μία αίτηση αφορά απομονωτή SRAM και το επισημαίνει στο υλικολογισμικό. Αν πρόκειται για αίτηση αποστολής, το υλικολογισμικό παραλείπει το αρχικό στάδιο DMA από τη RAM στην SRAM (μηχανή καταστάσεων SDMA), αν πρόκειται για λήψη, το υλικολογισμικό συλλέγει τα δεδομένα στην SRAM και ειδοποιεί μέσω της ουράς συμβάντων τη διεργασία-παραλήπτη.

Υποστήριξη πυρήνα Linux για απευθείας Ε/Ε με μνήμη PCI

Επεκτείνουμε το μηχανισμό εικονικής μνήμης του πυρήνα του ΛΣ, ώστε περιοχές μνήμης του περιφερειακού διαδρόμου να είναι κατάλληλες για συμμετοχή σε διαδικασίες απευθείας Ε/Ε (direct I/O). Μεταβάλλουμε το στάδιο αρχικοποίησης του συστήματος διαχείρισης μνήμης, ώστε να κατασκευάζει δομές διαχείρισης μνήμης (*πλαίσια* μνήμης) για το σύνολο του χώρου φυσικών διευθύνσεων (π.χ. 4GB για την αρχιτεκτονική i386), ανεξάρτητα από το διαθέσιμο ποσό RAM. Οι σχετικές δομές `struct page` ενσωματώνονται σε μια νέα ζώνη μνήμης, `ZONE_PCIMEM` και σημειώνονται ως δεσμευμένες.

Οι προτεινόμενες αλλαγές δεν είναι απλά μια *απεικόνιση* σε φυσικές διευθύνσεις· αφορούν στη *διαχείριση* των εν λόγω περιοχών από τον πυρήνα, οι οποίες πλέον αντιμετωπίζονται ως φυσική μνήμη του κόμβου. Οι λεπτομέρειες της υλοποίησης κρύβονται πίσω από την αφαίρεση του πλαισίου σελίδας· τα υπόλοιπα υποσυστήματα του πυρήνα δεν γνωρίζουν την ιδιαίτερη φύση των σελίδων αυτών.

0.3.3 Πειραματική αποτίμηση

Για την ποιοτική και ποσοτική αποτίμηση της βελτίωσης από τη χρήση του προτεινόμενου μονοπατιού, συγκρίνουμε πειραματικά τρία διαφορετικά συστήματα nbd: την πρότυπη υλοποίηση του gmblock με απομονωτές στην SRAM (εφεξής `gmblock-sram`), μια καθιερωμένη υλοποίηση πάνω από TCP/IP – το GNBD της Red Hat – και τέλος το gmblock χωρίς τις προτεινόμενες βελτιστοποιήσεις. Η επίδοσή του είναι αντιπροσωπευτική υλοποιήσεων RDMA με μονοπάτι που διασχίζει την κύρια μνήμη (`gmblock-ram`).

	Server A	Server B
Processor	2x Pentium III@1266MHz	Pentium 4@3GHz
Motherboard	Supermicro P3TDE6	Intel SE7210TP1-E
Chipset	Serverworks ServerSet III HE-SL, CIOB20 PCI bridge	Intel E7210 chipset, 6300ESB I/O controller hub
I/O Bus	2-slot 64bit/66MHz PCI	3-slot 64bit/66MHz PCI-X
RAM	2x PC133 512MB SDRAM	2 x PC2700 512MB SDRAM DDR
Disks	8x Western Digital WD2500JS 250GB SATA II	
I/O controller	3Ware 9500S-8 SATA RAID and MBL	
NIC	Myrinet M3F2-PCIXE-2	

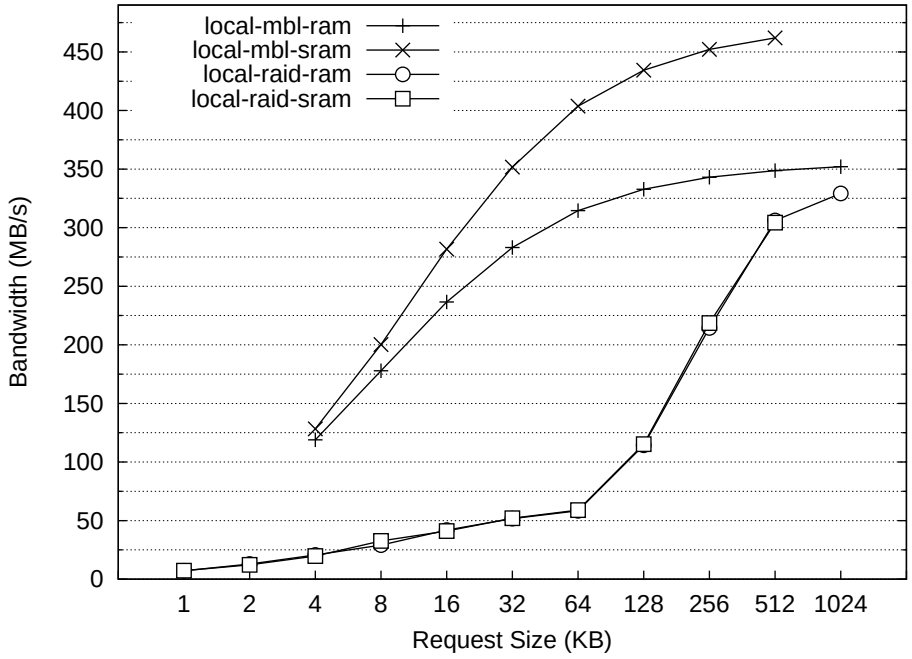
Πίνακας 1: *Προδιαγραφές υλικού των εξυπηρετητών αποθήκευσης*

Οι μετρικές που χρησιμοποιούμε είναι: *(a)* ρυθμός μεταφοράς για απομακρυσμένες αναγνώσεις δεδομένων *(b)* ρυθμός μεταφοράς για απομακρυσμένες εγγραφές δεδομένων *(c)* επιβάρυνση στην εκτέλεση τοπικών υπολογιστικών φορτίων στον εξυπηρετητή αποθήκευσης.

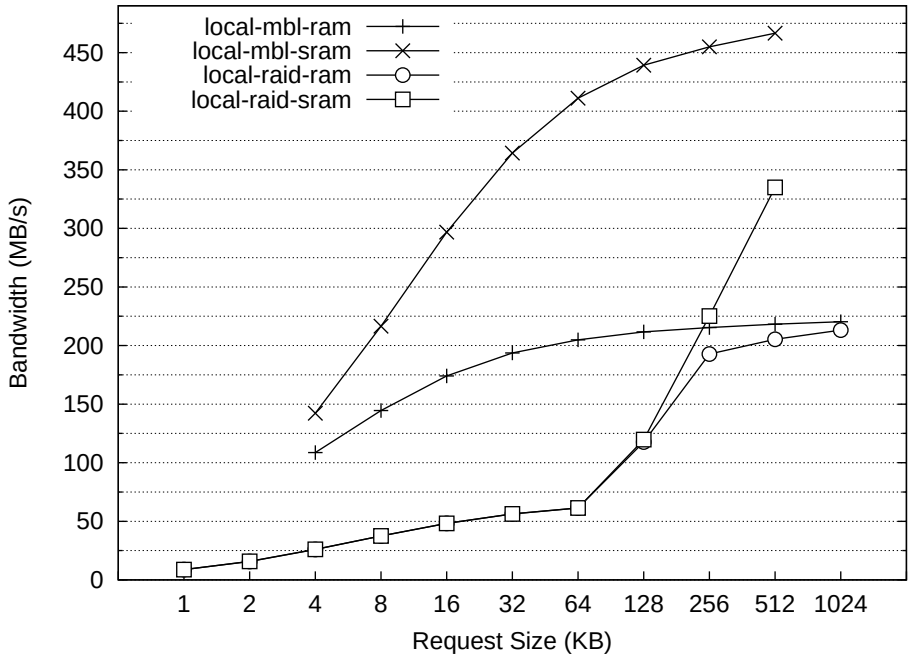
Χρησιμοποιούμε δύο είδη εξυπηρετητών (Σχ. 1). Το αποθηκευτικό μέσο είναι είτε μία διάταξη RAID0 που παρέχει ένας ελεγκτής 3Ware 9500S-8 είτε μια συσκευή αποθήκευσης στερεάς κατάστασης (solid-state disk) που υλοποιήσαμε με χρήση της μνήμης ενός δεύτερου προσαρμογέα Myrinet (Myrinet Block Device - MBL) και εξειδικευμένο υλικολογισμικό.

Πείραμα 1α: Επίδοση τοπικών μέσων

Μετράμε το τοπικά παρεχόμενο ρυθμό μεταφοράς από τα αποθηκευτικά μέσα για συνεχείς αιτήσεις μεταβλητού μήκους, όταν οι απομονωτές είναι είτε στη RAM είτε στην SRAM (Σχ. 14(b)). Ο ελεγκτής RAID έχει πολύ καλύτερη συμπεριφορά για μεγαλύτερες αιτήσεις, της τάξης των 128KB-256KB, καθώς αυτές εκτελούνται παράλληλα από περισσότερους δίσκους της διάταξης. Στην περίπτωση του εξυπηρετητή B, η απόδοση του `gmblock-ram` περιορίζεται από τη διασύνδεση του περιφερειακού διαδρόμου με την κύρια μνήμη, στα ∼217MB/s, λόγω περιορισμών του υλικού. Η γέφυρα PCI του εξυπηρετητή A εμφανίζει καλύτερη συμπεριφορά (Σχ. 14(a)).



(a) RAID και MBL, εξυπηρετητής A

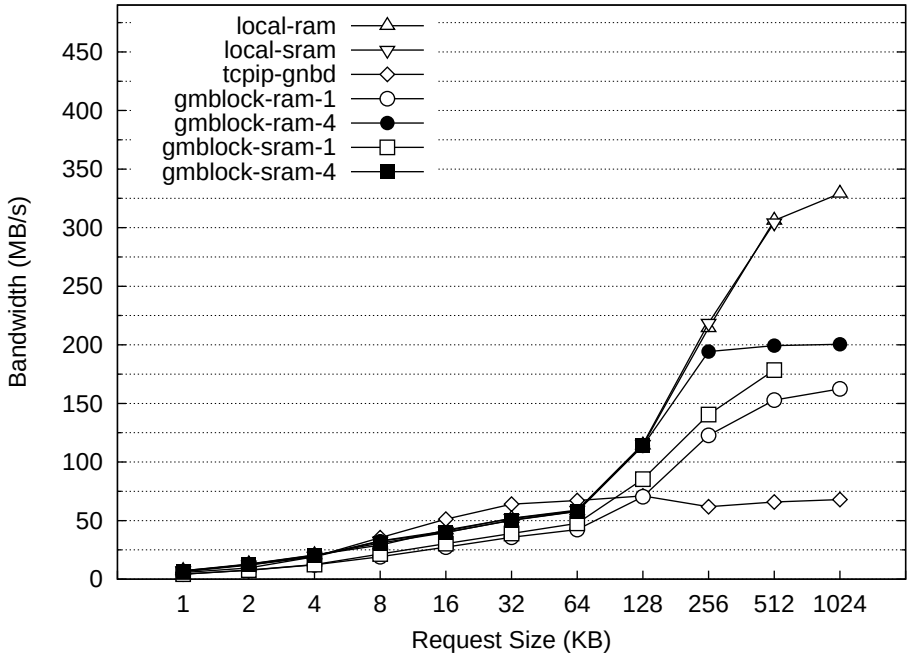


(b) RAID και MBL, εξυπηρετητής B

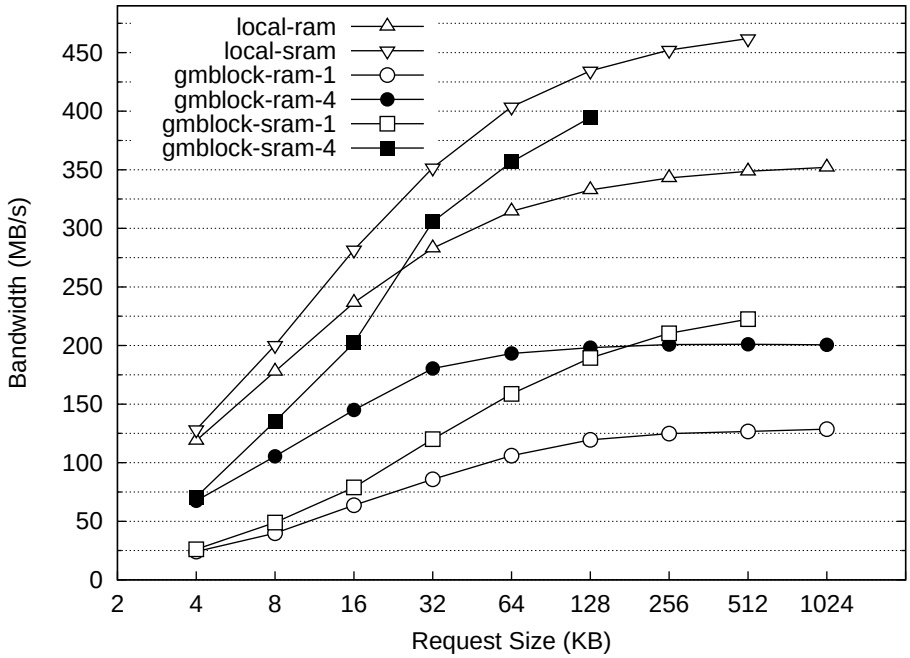
Σχήμα 14: *Τοπικός ρυθμός μεταφοράς, δύο αποθηκευτικά μέσα, εξυπηρετητές A και B*

Πείραμα 1β: Επίδοση απομακρυσμένων αναγνώσεων

Μετράμε το ρυθμό εξυπηρέτησης για αιτήσεις απομακρυσμένης ανάγνωσης, με μία, δύο ή τέσσερις αιτήσεις σε εξέλιξη για αύξηση του βαθμού χρησιμοποίησης του δι-



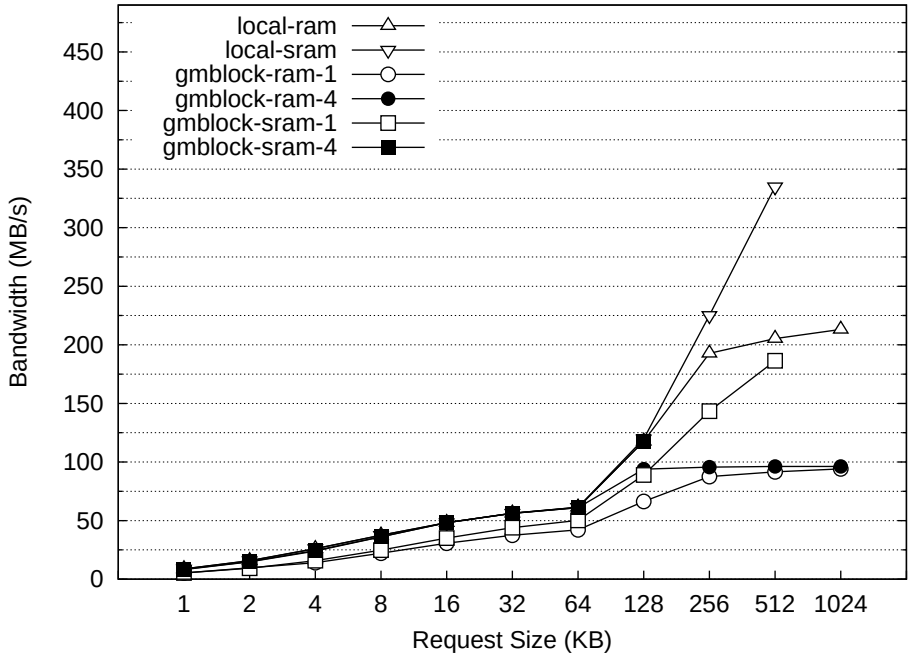
(a) Ρυθμός μεταφοράς, RAID



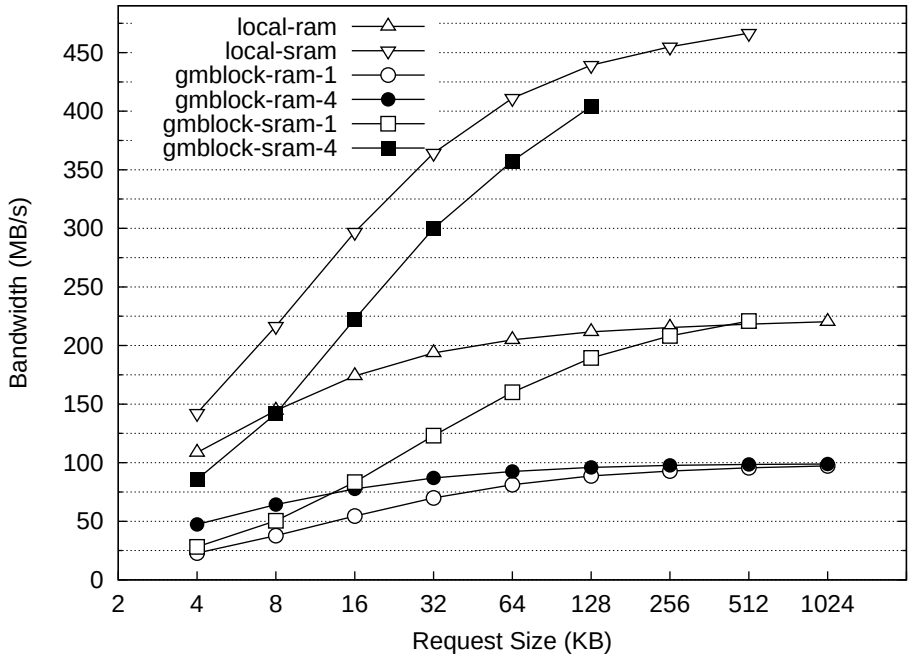
(b) Ρυθμός μεταφοράς, MBL

Σχήμα 15: *Ρυθμός μεταφοράς απομακρυσμένων αναγνώσεων, εξυπηρετητής A*

κτυακού συνδέσμου (διατάξεις `gmblock-ram-{1,2,4}` και `gmblock-sram-{1,2,4}`). Για λόγους απλότητας των σχημάτων, παραλείπουμε τις καμπύλες για τα `gmblock-{ram,sram}-2`.



(a) Ρυθμός μεταφοράς, RAID



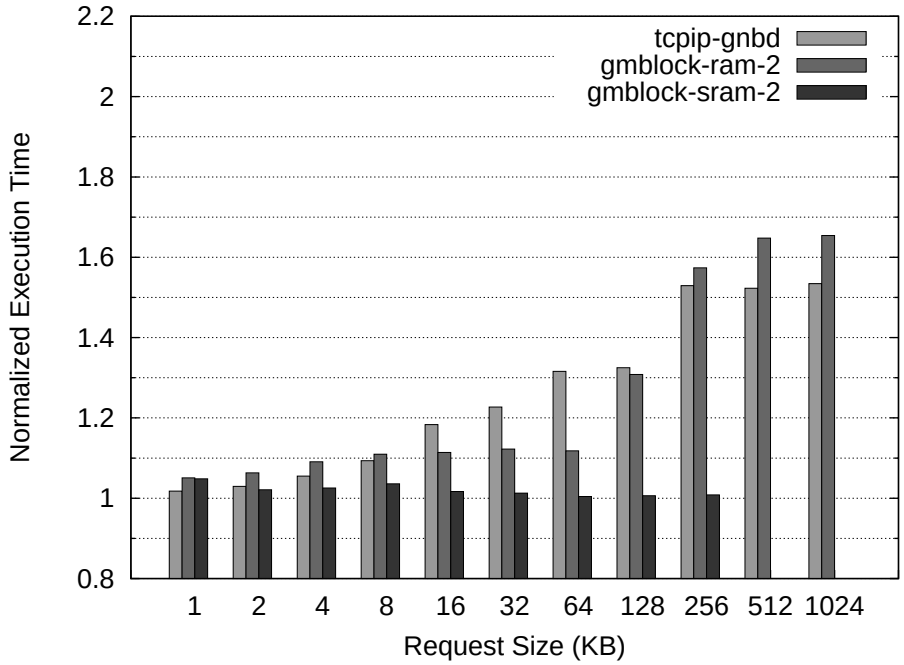
(b) Ρυθμός μεταφοράς, MBL

Σχήμα 16: *Ρυθμός μεταφοράς απομακρυσμένων αναγνώσεων, εξυπηρετητής B*

Το GNBD έχει πολύ χαμηλή απόδοση, λόγω του υψηλού κόστους επεξεργασίας του πρωτοκόλλου TCP/IP και κορεσμού του διαδρόμου κύριας μνήμης (Σχ. 15(a)).

Η απόδοση των `gmblock-ram-1` και `gmblock-sram-1` καθορίζεται από το χρόνο αρχι-

κής απόκρισης γιατί μόνο μία αίτηση είναι σε εξέλιξη, οπότε ο βαθμός χρησιμοποίησης των πόρων είναι χαμηλός. Με δύο ή τέσσερις αιτήσεις σε εξέλιξη, βλέπουμε ότι το εύρος ζώνης του συνδέσμου από τον περιφερειακό διάδρομο προς τη μνήμη γίνεται η στενωπός του συστήματος και καθορίζει τη συνολική επίδοση του `gmblock-ram`. Από την άλλη πλευρά, το `gmblock-sram` δεν εμφανίζει ανάλογο περιορισμό, με τα `gmblock-sram-{2,4}` να επιτυγχάνουν σχεδόν το 90% του τοπικού ρυθμού μεταφοράς για αιτήσεις των 256KB και 128KB αντίστοιχα, σχεδόν δύο φορές καλύτερα από το `gmblock-ram`.



Σχήμα 17: *Ανταγωνισμός στο διάδρομο κύριας μνήμης*

Η διαφορά θα ήταν περισσότερο εμφανής για μεγαλύτερα μεγέθη αιτήσεων, ωστόσο το μέγεθος της διαθέσιμης μνήμης στον προσαρμογέα δικτύου περιορίζει τον αριθμό των αιτήσεων σε εξέλιξη και το μέγεθός τους. Στη συνέχεια, προτείνουμε τις συγχρονισμένες λειτουργίες αποστολής ως ένα μέσο για να αρθεί αυτός ο περιορισμός (§ 0.4).

Πείραμα 1γ: Επίδραση στον τοπικό υπολογισμό

Το Σχ. 17 παρουσιάζει την επιβάρυνση της απομακρυσμένης Ε/Ε σε τοπικό υπολογισμό που εξελίσσεται παράλληλα στον εξυπηρετητή αποθήκευσης. Παρόλο που το `gmblock-ram` αφαιρεί τη CPU από το κρίσιμο μονοπάτι, συνεχίζει να καταναλώνει εύ-

ρος ζώνης στο διάδρομο μνήμης. Τρέξαμε τα `tcpip-gnbd`, `gmblock-ram-2` και `gmblock-sram-2` μαζί με ένα υπολογιστικά απαιτητικό πρόγραμμα, μια διεργασία του προγράμματος συμπίεσης `bzip2` σε έναν από τους επεξεργαστές του εξυπηρετητή A. Παρόλο που δεν υπάρχει διαμοιρασμός υπολογιστικού χρόνου, στη χειρότερη περίπτωση το `bzip2` τρέχει 67% αργότερα, όταν εκτελείται παράλληλα με το `gmblock-ram-2` και αιτήσεις των 512KB, ενώ η επιβάρυνσή του είναι αμελητέα όταν εκτελείται παράλληλα με το `gmblock-sram`.

Πείραμα 1δ: Επίδοση απομακρυσμένων εγγραφών

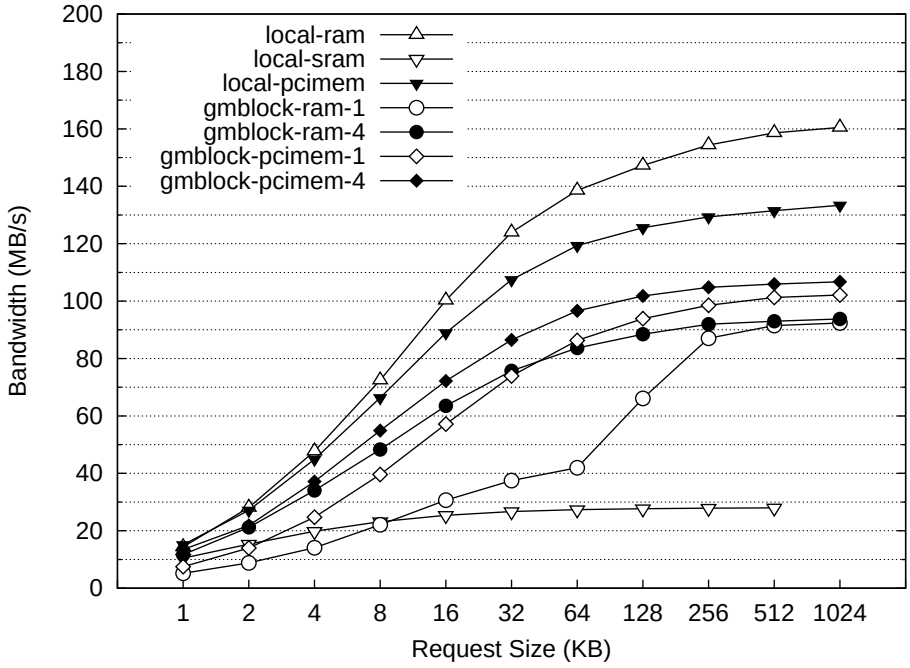
Αποτιμούμε την επίδοση λειτουργιών απομακρυσμένης εγγραφής δεδομένων· τα αποτελέσματα απεικονίζονται στο Σχ. 18. Το `gmblock-ram` και πάλι περιορίζεται λόγω της διάσχισης του διαδρόμου μνήμης. Ωστόσο, η επίδοση του `gmblock-sram` είναι σημαντικά χαμηλότερη από τα αναμενόμενα. Ανακαλύψαμε και επιβεβαιώσαμε με τη Myricom ότι αυτό οφείλεται σε αρχιτεκτονικό περιορισμό του επεξεργαστή LanaiX, που δεν έχει ικανοποιητική απόδοση ως στόχος δοσοληψιών ανάγνωσης από το PCI.

Για να παρακάμψουμε τον περιορισμό αυτό εισάγουμε ένα ακόμη μονοπάτι (`gmblock-pcimem`), το οποίο χρησιμοποιεί ενδιάμεση μνήμη επάνω στο PCI. Οι απομονωτές παρέχονται από έναν προσαρμογέα βασισμένο σε επεξεργαστή Intel XScale, τον Cyclone 740 [Cyc].

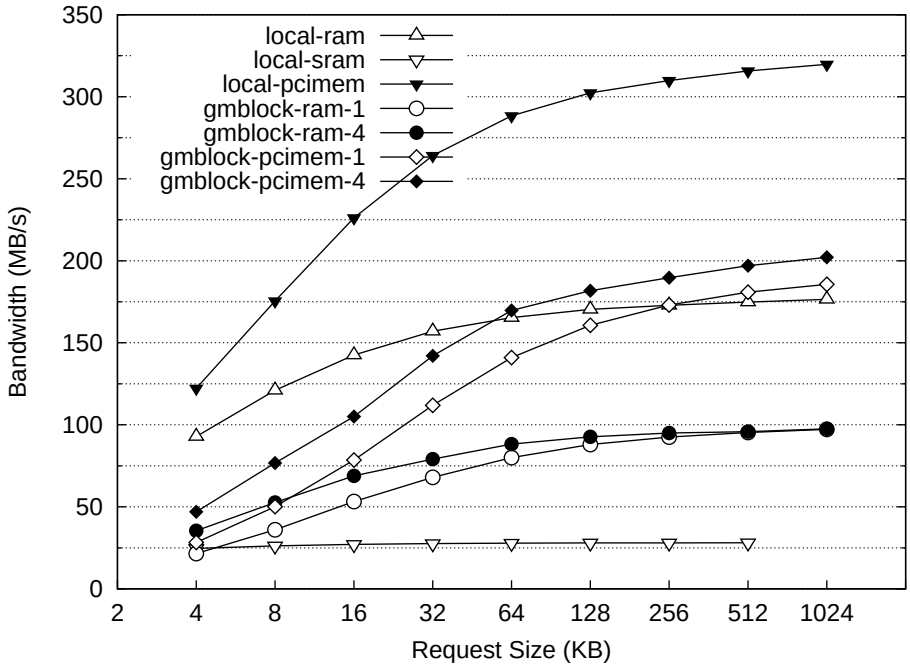
0.3.4 Συζήτηση

Πρόσβαση σε δομημένα δεδομένα

Το προτεινόμενο μονοπάτι δεδομένων φαίνεται στο Σχ. 12(a). Εφόσον η CPU εμπλέκεται κατά την εγκατάσταση του μονοπατιού, στη φάση αρχικοποίησης της μεταφοράς, το gmblock μπορεί να χρησιμοποιηθεί για το διαμοιρασμό *δομημένων* δεδομένων. Το προτεινόμενο σύστημα εστιάζει στην *κίνηση* των δεδομένων, απομπλέκοντας το μονοπάτι ελέγχου από το μονοπάτι δεδομένων, χωρίς να θέτει περιορισμούς στην οργάνωσή τους. Θα μπορούσε για παράδειγμα να χρησιμοποιηθεί για το διαμοιρασμό αρχείων από ένα καθιερωμένο σύστημα `ext2`, π.χ. για χρήση ως δίσκους σε επίπεδο μπλοκ από εικονικές μηχανές.



(a) Ρυθμός μεταφοράς, RAID



(b) Ρυθμός μεταφοράς, MBL

Σχήμα 18: *Ρυθμός μεταφοράς απομακρυσμένων εγγραφών, εξυπηρετητής B*

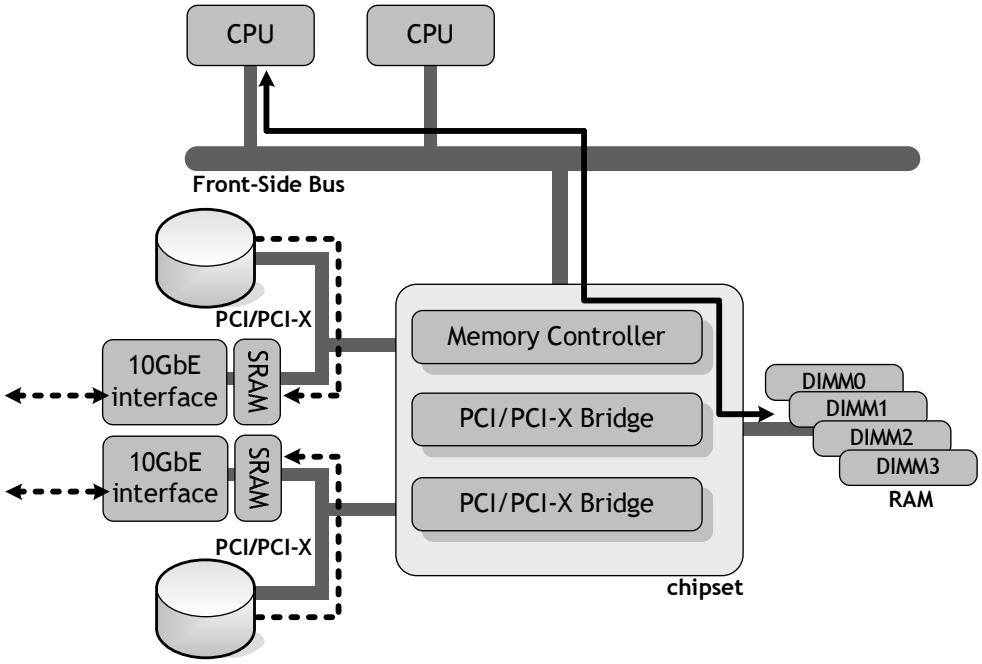
Επιπλέον, η χρήση του μηχανισμού `O_DIRECT` εξασφαλίζει συνάφεια με την κρυφή μνήμη σελίδων του πυρήνα καθώς τα δεδομένα μετακινούνται ανάμεσα στο μέσο και το δίκτυο. Ο πυρήνας παραμένει στο στάδιο αρχικοποίησης μεταφοράς, οπότε φροντίζει

να γράφει πίσω στο δίσκο δεδομένα που βρίσκονται στην κρυφή μνήμη και ζητούνται για απομακρυσμένη ανάγνωση και αντίστοιχα να ακυρώνει δεδομένα που βρίσκονται στην κρυφή μνήμη όταν τα αντίστοιχα μπλοκ ανανεώνονται μέσω απομακρυσμένης εγγραφής.

Κρυφή μνήμη και προανάκτηση δεδομένων

Η παράκαμψη της κύριας μνήμης του εξυπηρετητή αποθήκευσης αφαιρεί τη δυνατότητα για χρήση κρυφής μνήμης και προανάκτηση δεδομένων (prefetching) στην πλευρά του εξυπηρετητή. Η προανάκτηση συνεχίζει να είναι δυνατή, αλλά πρέπει να εκκινείται από την πλευρά του πελάτη (βλ. § 0.5.3). Προσωρινή αποθήκευση δεδομένων γίνεται επίσης στην πλευρά των πελατών, οι οποίοι αντιμετωπίζουν το μοιραζόμενο χώρο ως τοπικό δίσκο· το συνολικό μέγεθος μνήμης των πελατών περιμένουμε να ξεπερνά κατά πολύ τη διαθέσιμη μνήμη του εξυπηρετητή αποθήκευσης. Η προσωρινή αποθήκευση στον εξυπηρετητή μπορεί να έχει θετική επίδραση σε εφαρμογές που εκτελούν συχνά εγγραφές και αναγνώσεις σε κοινά δεδομένα, με τον εξυπηρετητή να αποτελεί ουσιαστικά έναν ενδιάμεσο απομονωτή για μεταφορές από πελάτη σε πελάτη. Για τέτοια φορτία, η χρήση του σύντομου μονοπατιού του gmblock δεν είναι κατάλληλη επιλογή.

Από την άλλη πλευρά, η επίπτωση της έντονης κοινής χρήσης δεδομένων για εγγραφή κι ανάγνωση αποτελεί γνωστό πρόβλημα, για την αντιμετώπιση του οποίου έχουν προταθεί τεχνικές που χρησιμοποιούνται σήμερα σε συστήματα παραγωγής, τόσο στο επίπεδο της εφαρμογής όσο και στο επίπεδο του συστήματος αρχείων. Η βάση Oracle 9i υποστηρίζει τον μηχανισμό κατανεμημένης κρυφής μνήμης «Cache Fusion» [LSC+01] ο οποίος επιτρέπει απευθείας μεταφορές δεδομένων από πελάτη σε πελάτη κι εκτελεί Ε/Ε με το αποθηκευτικό μέσο μόνο όταν τα απαιτούμενα δεδομένα δεν βρίσκονται προσωρινά αποθηκευμένα σε *κανέναν* από τους συμμετέχοντες κόμβους. Ομοίως, για εφαρμογές MPI με έντονη κοινή χρήση αρχείων το GPFS προσφέρει τη δυνατότητα «data shipping» [PTH+01, BICrT08], η οποία μεταφέρει δεδομένα απευθείας ανάμεσα σε διεργασίες, ώστε μόνο μία να εκτελεί Ε/Ε σε συγκεκριμένο τμήμα του κοινού αρχείου κατά την εκτέλεση συλλογικής Ε/Ε.



Σχήμα 19: *SRAM ανάμεσα στον περιφερειακό διάδρομο και το δίκτυο*

Εφαρμογή σε εναλλακτικές τεχνολογίες δικτύωσης

Η υλοποίηση του gmblock βασίζεται στο Myrinet, το οποίο προσφέρει μια πλήρως προγραμματιζόμενη κάρτα δικτύου. Τέτοια δυνατότητα αυξάνει την ευελιξία του συστήματος, ωστόσο δεν είναι απαραίτητη για την υλοποίηση του προτεινόμενου μονοπατιού. Οι προϋποθέσεις για την κατασκευή του μονοπατιού είναι η χρήση αποθηκευτικού μέσου και δικτύου διασύνδεσης με δυνατότητα DMA καθώς και η ύπαρξη ενός μικρού ποσού μνήμης, προσβάσιμο μέσω του χώρου διευθύνσεων του PCI, *κοντά* στο δίκτυο. Έτσι, για να είναι η προσέγγισή μας εφαρμόσιμη σε εναλλακτικές τεχνολογίες δικτύωσης, όπως το Infiniband και το 10-Gigabit Ethernet, προτείνουμε την προσθήκη ενός μικρού ποσού μνήμης SRAM ανάμεσα στον περιφερειακό διάδρομο και τη διεπαφή του δικτύου. Το Σχ. 19 απεικονίζει μια τέτοια διάταξη.

Ο επεξεργαστής του κόμβου εκκινεί μια δοσοληψία DMA από το αποθηκευτικό μέσο στο εύρος διευθύνσεων PCI που αντιστοιχούν στην SRAM. Η εμπλοκή του στη συνέχεια εξαρτάται από το είδος του δικτύου· στην περίπτωση του TCP/IP τρέχει η στοίβα πρωτοκόλλων αλλά η κάρτα δικτύου συλλέγει δεδομένα πακέτων από την SRAM μέσω DMA, ενώ στην περίπτωση του Infiniband δηλώνονται οι αντίστοιχες περιοχές της SRAM για χρήση με λειτουργίες RDMA, τις οποίες αναλαμβάνει ανεξάρτητα ο προσαρμογέας δικτύου.

Εφαρμογή σε σχεδιάσεις χαμηλής κατανάλωσης ενέργειας

Η προσέγγιση του gmblock είναι σχετική με την τάση για εξυπηρετητές αποθήκευσης χαμηλής κατανάλωσης ενέργειας, βασισμένους σε αρχιτεκτονικές συστήματος σε ψηφίδα (System-on-Chip) [Mar]. Οι συσκευές αυτές χρησιμοποιούν συνήθως ρολόγια της τάξης των 500MHz-1GHz και εσωτερικούς διαδρόμους χαμηλού εύρους ζώνης για να κρατηθεί χαμηλά το κόστος κατασκευής και η κατανάλωση ενέργειας. Η χρήση του προτεινόμενου μονοπατιού αντί για ενδιάμεση αποθήκευση σε κύρια μνήμη SDRAM ή DDR θα μείωνε τον ανταγωνισμό για πρόσβαση· η κύρια μνήμη θα ήταν διαθέσιμη για επεξεργασία του δικτυακού πρωτοκόλλου στη CPU, βελτιώνοντας τη δυνατότητα κλιμάκωσης του συστήματος σε περισσότερους δίσκους και γρηγορότερα δίκτυα διασύνδεσης. Επιπλέον, η χρήση περιοχών SRAM κατάλληλου μεγέθους μπορεί να επιφέρει σημαντική βελτίωση της κατανάλωσης ενέργειας [MACM05, ACMM07]. Η μείωση του φόρτου στην κύρια μνήμη επιτρέπει την εφαρμογή επιθετικών τεχνικών εξοικονόμησης ενέργειας: μια διάταξη DDR266 χαμηλού ρυθμού μεταφοράς καταναλώνει λιγότερο από 17% της απαιτούμενης ισχύος για μια διάταξη DDR333 υψηλού ρυθμού μεταφοράς [Jan01].

0.4 Συγχρονισμένες λειτουργίες αποστολής GM

0.4.1 Κίνητρο

Παρόλο που το gmblock επιτυγχάνει σημαντική βελτίωση στο ρυθμό μεταφοράς δεδομένων για απομακρυσμένες λειτουργίες ανάγνωσης/εγγραφής, η επίδοσή του δεν φτάνει στο όριο των δυνατοτήτων του αποθηκευτικού μέσου και του δικτύου διασύνδεσης. Αυτό οφείλεται στην αλληλεπίδραση της συμπεριφοράς της διάταξης RAID, που απαιτεί αιτήσεις μεγάλου μήκους για να παρέχει αρκετά υψηλό ρυθμό μεταφοράς, με το περιορισμένο ποσό μνήμης που διαθέτει ο προσαρμογέας δικτύου, το οποίο δεν επιτρέπει μεγάλο αριθμό αιτήσεων ικανοποιητικού μεγέθους να είναι σε εξέλιξη ταυτόχρονα. Στην πειραματική μας διάταξη δεν μπορέσαμε να δεσμεύσουμε πάνω από 700KB μνήμης στον προσαρμογέα για χρήση του gmblock, οπότε ο αριθμός των αιτήσεων που μπορούν να είναι σε εξέλιξη είναι το πολύ το πολύ 1×512KB, 2×256KB, ή 4×128KB. Χρησιμοποιώντας πολλές μικρότερες αιτήσεις έχουμε καλύτερη χρησιμοποί-

ηση των διαφορετικών σταδίων επεξεργασίας του συστήματος (καλύτερο pipelining), αλλά μειώνεται η απόδοση της διάταξης RAID. Επιπλέον, αυξάνεται η επιβάρυνση της επεξεργασίας, εφόσον το σύστημα έχει να διαχειριστεί μεγαλύτερο αριθμό αιτήσεων για δεδομένο ρυθμό μεταφοράς.

Για να δώσουμε τη δυνατότητα χρήσης μεγαλύτερων αιτήσεων με μικρή επιβάρυνση και καλό βαθμό χρησιμοποίησης του αποθηκευτικού μέσου και του δικτύου, εστιάζουμε σε τρόπους επικάλυψης διαφορετικών φάσεων επεξεργασίας για την *ίδια* αίτηση. Ο στόχος είναι μια αίτηση ανάγνωσης να μπορεί να περάσει στο στάδιο δικτυακής αποστολής πριν ακόμη να έχει ολοκληρωθεί η εξυπηρέτησή της από το δικτυακό μέσο, έτσι ώστε να εκμεταλλευόμαστε πλήρως τις δυνατότητες του αποθηκευτικού μέσου ακόμη και με μικρό αριθμό αιτήσεων σε εξέλιξη.

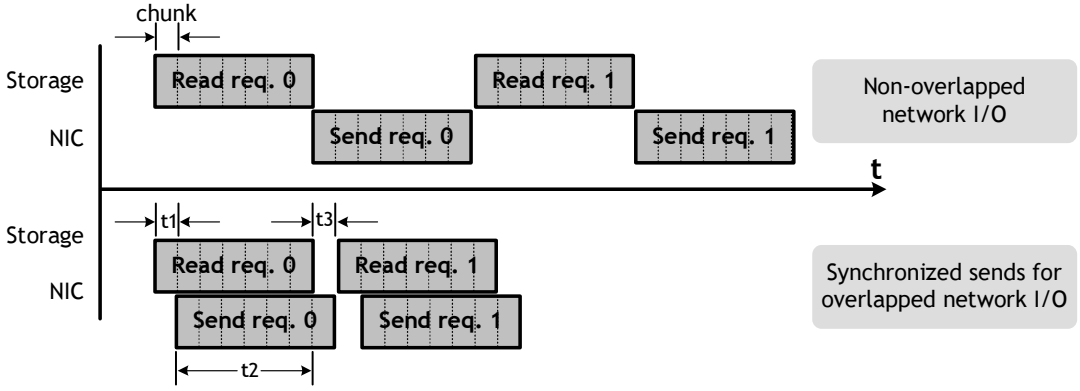
0.4.2 Σχεδίαση

Η εξυπηρέτηση μιας αίτησης ανάγνωσης μπλοκ περιλαμβάνει δύο βήματα: την ανάκτηση των δεδομένων από το αποθηκευτικό μέσο στη μνήμη του προσαρμογέα δικτύου και την αποστολή δικτυακών πακέτων από τη μνήμη του προσαρμογέα στο φυσικό μέσο. Για να γίνει δυνατή η επικάλυψη των φάσεων μέσα στην ίδια αίτηση, χρειάζεται να *συγχρονιστεί* η δικτυακή αποστολή με τη διαδικασία ανάγνωσης από το δίσκο, εξασφαλίζοντας ότι μόνο έγκυρα δεδομένα αποστέλλονται στο δίκτυο. Ιδανικά, η διαδικασία πρέπει να εισάγει μικρή επιβάρυνση, να είναι ανεξάρτητη της αποθηκευτικής συσκευής και να είναι συμβατή με τη σημασιολογία των κλήσεων που χρησιμοποιούνται από τον εξυπηρετητή nbd για Ε/Ε από το δίσκο και δικτυακή επικοινωνία.

Μια τέτοια προσέγγιση θα μπορούσε να υλοποιηθεί αμιγώς σε λογισμικό: ο εξυπηρετητής διαχειρίζεται μεγάλες αιτήσεις, τις οποίες διασπά σε τμήματα για την επεξεργασία τους από το δίκτυο. Ο εξυπηρετητής μπορεί να διαιρεί κάθε μεγαλύτερη αίτηση ανάγνωσης των l bytes (π.χ. 1MB) σε μικρότερα τμήματα των c bytes (π.χ. 4KB), τα οποία υποβάλλει προς επεξεργασία από το αποθηκευτικό μέσο. Ωστόσο, αυτή η σχεδίαση έχει σημαντικά μειονεκτήματα: αυξάνει το ρυθμό των δικτυακών μηνυμάτων, εισάγει μεγάλο κόστος επεξεργασίας των αιτήσεων, εφόσον και ο εξυπηρετητής και ο πελάτης έχουν να διαχειριστούν υψηλό ρυθμό συμβάντων ολοκλήρωσης αιτήσεων Ε/Ε από το δίσκο και το δίκτυο κι αγνοεί το γεγονός ότι οι μικρότερες αιτήσεις αφορούν συνεχόμενα τμήματα στο αποθηκευτικό μέσο.

Προτείνουμε ένα μηχανισμό συγχρονισμού ο οποίος λειτουργεί απευθείας ανάμεσα στη συσκευή αποθήκευσης και ένα προγραμματιζόμενο προσαρμογέα δικτύου, όπως του Myrinet, με τρόπο που παρακάμπτει εντελώς την CPU του κόμβου και το ΛΣ που εκτελείται σε αυτή, παραμένοντας ανεξάρτητος του είδους της αποθηκευτικής συσκευής.

Ας θεωρήσουμε το σενάριο όπου ο εξυπηρετητής ξεκινά μια λειτουργία αποστολής δεδομένων στο δίκτυο πριν από την κλήση συστήματος `read()` προς το στρώμα Ε/Ε του ΛΣ. Στην περίπτωση αυτή, είναι σίγουρο ότι οι δύο φάσεις επικαλύπτονται, ωστόσο το σύστημα πιθανότατα θα αποτύχει, γιατί δεν εξασφαλίζεται η ορθότητα των δεδομένων που εισάγονται στο δίκτυο. Για να λύσουμε το πρόβλημα, εισάγουμε την ιδιότητα του *συγχρονισμού* σε λειτουργίες αποστολής επιπέδου χρήστη. Μια συγχρονισμένη λειτουργία του GM εξασφαλίζει ότι τα δεδομένα που πρόκειται να αποσταλούν από έναν απομονωτή είναι έγκυρα πριν την κατασκευή των αντίστοιχων δικτυακών πακέτων. Αν κάποια στιγμή δεν υπάρχουν έγκυρα δεδομένα, ο προσαρμογέας παραλείπει την κατασκευή πακέτων για το συγκεκριμένο αίτημα αποστολής.

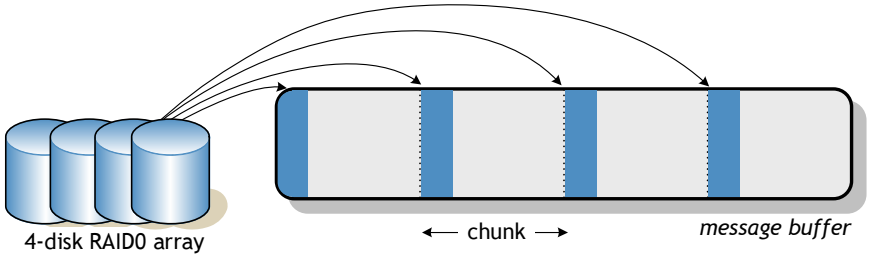


Σχήμα 20: *Επικάλυψη φάσεων για την ίδια αίτηση Ε/Ε*

Ο προσαρμογέας εντοπίζει την ολοκλήρωση της μεταφοράς δεδομένων σε τμήματα των c bytes. Η τιμή του c επηρεάζει το βαθμό επικάλυψης (Σχ. 20). Το αποθηκευτικό μέσο μεταφέρει δεδομένα για τις πρώτες $t_1 = \frac{c}{r_{disk}}$ χρονικές μονάδες, στη συνέχεια για $t_2 = \frac{l-c}{r_{disk}}$ και η κάρτα δικτύου και το αποθηκευτικό μέσο είναι ενεργά και τέλος για $t_3 = \frac{c}{r_{net}}$ μόνο η κάρτα δικτύου μεταφέρει δεδομένα. Μικρότερες τιμές του c οδηγούν σε καλύτερη επικάλυψη, αλλά αυξάνουν το φόρτο στο Lanai.

Οι συγχρονισμένες λειτουργίες αποστολής αίρουν τον περιορισμό ότι όλα τα δεδομένα πρέπει να είναι διαθέσιμα πριν από την εκκίνηση μιας δικτυακής αποστολής. Ωστόσο, όπως είδαμε πειραματικά και περιγράφεται αναλυτικότερα στη συνέχεια (§ 0.4.4), αυτό

δεν αρκεί για την επίτευξη καλής επίδοσης στην πειραματική μας πλατφόρμα. Η σχεδίαση υποθέτει ότι τα δεδομένα γράφονται σειριακά στον απομονωτή, ωστόσο αυτό δεν ισχύει για αποθηκευτικές διατάξεις RAID (Σχ. 21.



Σχήμα 21: *Ανάγνωση λωρίδας δεδομένων από διάταξη RAID0 τεσσάρων δίσκων*

Για να επιτύχουμε καλό βαθμό επικάλυψης, χρειάζεται να προσαρμόσουμε τις συγχρονισμένες λειτουργίες αποστολής στην ύπαρξη πολλών διαφορετικών εισερχόμενων ρευμάτων δεδομένων. Για το λόγο αυτό επεκτείνουμε τη λειτουργία της αποστολής στο GM ώστε ο αποστολέας να έχει τη δυνατότητα να παράγει πακέτα από οποιοδήποτε σημείο του απομονωτή. Αντίστοιχα, ο παραλήπτης επεκτείνεται ώστε να υποστηρίζει εκτός σειράς τοποθέτηση των εισερχόμενων τεμαχίων του μηνύματος.

Επειδή το κόστος του εντοπισμού εισερχόμενων δοσοληψιών DMA οπουδήποτε μέσα στον απομονωτή του μηνύματος είναι απαγορευτικό, υποστηρίζουμε συγχρονισμένες λειτουργίες πολλαπλών ρευμάτων ζητώντας από τον χρήστη να υποδεικνύει τις αρχικές θέσεις και το μήκος του κάθε ρεύματος δεδομένων μέσα στον απομονωτή. Στην περίπτωσή μας, οι τιμές προκύπτουν από το πλήθος των δίσκων που συμμετέχουν στη διάταξη RAID και το μήκος της λωρίδας αποθήκευσης.

Παρόλο που η πρότυπη υλοποίηση βασίζεται στο Myrinet/GM, η υλοποίηση συγχρονισμένων λειτουργιών αποστολής μπορεί να μεταφερθεί σε οποιοδήποτε δίκτυο εξάγει τμήμα της μνήμης του στο χώρο διευθύνσεων του PCI και παρέχει τον αναγκαίο βαθμό προγραμματισιμότητας στον προσαρμογέα δικτύου. Ο συγχρονισμός γίνεται απευθείας πάνω από τον περιφερειακό διάδρομο, χωρίς εμπλοκή της CPU.

0.4.3 Θέματα υλοποίησης στο Myrinet/GM

Το κυριότερο θέμα υλοποίησης που προκύπτει είναι πώς ο Lanai θα ενημερώνεται για την ολοκλήρωση δοσοληψιών DMA από εξωτερικό παράγοντα προς τη μνήμη του. Ο

προσαρμογέας Myrinet δεν παρέχει υποστήριξη στο υλικό για τέτοια λειτουργικότητα, για παράδειγμα έναν πίνακα από bits που θα υποδείκνυαν ότι το αντίστοιχο τμήμα της SRAM μεταβλήθηκε.

Εφόσον δεν υπάρχει υποστήριξη από το υλικό, υλοποιούμε αντίστοιχη λειτουργικότητα στο υλικολογισμικό: ο Lanai γράφει γνωστές τιμές των 32 bits σε τακτά διαστήματα στη μνήμη και εντοπίζει πότε αυτές οι τιμές αλλοιώνονται, οπότε ολοκληρώθηκε μια λειτουργία DMA στο αντίστοιχο διάστημα. Θεωρούμε ότι η εγγραφή γίνεται σειριακά μέσα σε κάθε τμήμα. Η πιθανότητα να χαθεί μια αλλαγή λόγω ταύτισης της τιμής που γράφεται με την αρχική τιμή, για τυχαία δεδομένα και l = 1MB, c = 4KB, είναι:

$$P_{no\_ovr} = 1 - \left(1 - 2^{-32}\right)^{\lceil \frac{l}{c} \rceil} \Longrightarrow P = 5.96 \times 10^{-8}$$

.

Για να εξασφαλιστεί ότι η διαδικασία ολοκληρώνεται σε αυτή την περίπτωση, υπάρχει ένα μια ακόμη χαρακτηριστική τιμή, μετά το τέλος του απομονωτή του μηνύματος, που τίθεται από τον εξυπηρετητή nbd όταν το σύνολο των δεδομένων έχει μεταφερθεί. Στην χειρότερη περίπτωση, με πιθανότητα P_{no_ovr} οι φάσεις ανάγνωσης και αποστολής δεν επικαλύπτονται πλήρως.

Η υλοποίηση των συγχρονισμένων λειτουργιών στο Myrinet/GM περιλαμβάνει τρία στάδια. Στο στάδιο αρχικοποίησης σημειώνονται οι αρχικές τιμές ευθυγραμμισμένες με πακέτα του GM μέσα στον απομονωτή και καλείται η `gm_synchro_send_with_callback()`. Προαιρετικά, ο χρήστης παρέχει υποδείξεις για την αρχή επιμέρους ρευμάτων δεδομένων μέσα στον απομονωτή. Στο στάδιο μετάδοσης, που εκτελείται κάθε φορά που πρόκειται να κατασκευαστεί πακέτο για αποστολή στο δίκτυο, η μηχανή SDMA του υλικολογισμικού εξετάζει ένα-ένα τα εισερχόμενα ρεύματα δεδομένων μέχρι να βρει κάποιο με έγκυρα δεδομένα. Τέλος, στο στάδιο ολοκλήρωσης της μεταφοράς, ο εξυπηρετητής nbd ενημερώνει το υλικολογισμικό ότι όλα τα δεδομένα είναι πλέον έγκυρα.

Για την υποστήριξη πολλαπλών ρευμάτων τροποποιήσαμε το πρωτόκολλο Go Back N του GM έτσι ώστε να υποστηρίζει την κατασκευή πακέτων στον αποστολέα και τοποθέτηση πακέτων στον παραλήπτη εκτός σειράς. Για κάθε πακέτο που κατασκευάζεται, ο αποστολέας κρατά το ρεύμα από το οποίο προήλθε, ώστε να μπορεί να το επανα-

φέρει σε περίπτωση αρνητικής επιβεβαίωσης. Αντίστοιχα, ο παραλήπτης χρησιμοποιεί ένα επιπλέον πεδίο `h_synchro_ptr` που προσθέσαμε στην κεφαλίδα των πακέτων του GM, ώστε να καθορίσει την τελική θέση του στην περιοχή προορισμού.

0.4.4 Πειραματική αποτίμηση

Το τμήμα αυτό παρουσιάζει μια πειραματική σύγκριση ανάμεσα στη βασική έκδοση του gmblock και την υλοποίηση που έχει επεκταθεί ώστε να υποστηρίζει συγχρονισμένες λειτουργίες. Η σύγκριση γίνεται σε εξυπηρετητή τύπου Β, καθώς διαθέτει γρηγορότερο περιφερειακό διάδρομο.

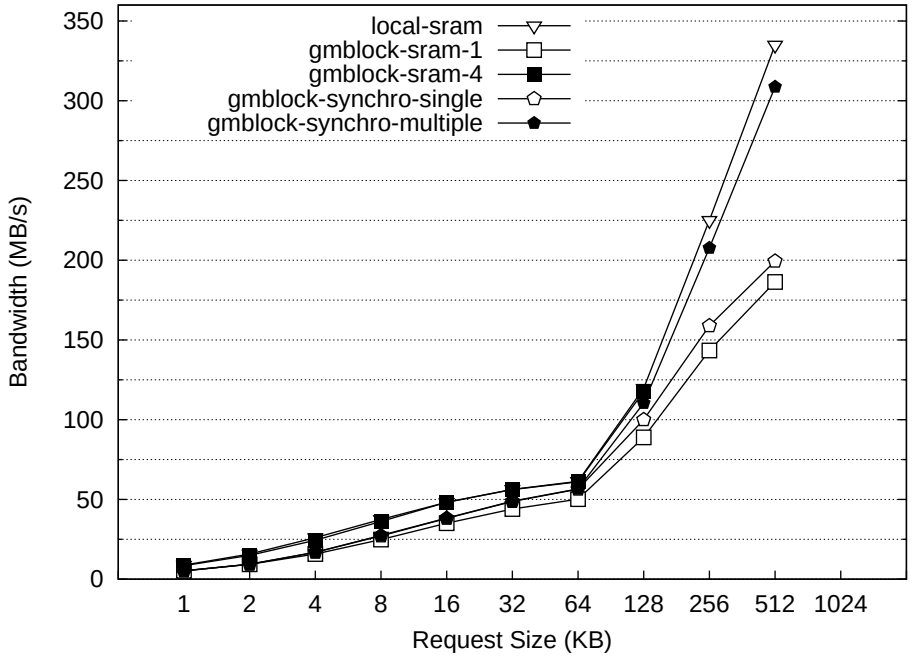
Πείραμα 2α: Συγχρονισμένες λειτουργίες αποστολής

Η έκδοση `gmblock-synchro-single` χρησιμοποιεί συγχρονισμένες λειτουργίες για ένα ρεύμα δεδομένων, με μία αίτηση σε εξέλιξη, ενώ η έκδοση `gmblock-synchro-multiple` ορίζει πολλαπλά ρεύματα εισερχόμενων δεδομένων, όσα και ο αριθμός των δίσκων στη διάταξη RAID. Τα αποτελέσματα φαίνονται στα Σχ. 22(a), 22(b) για τα δύο αποθηκευτικά μέσα.

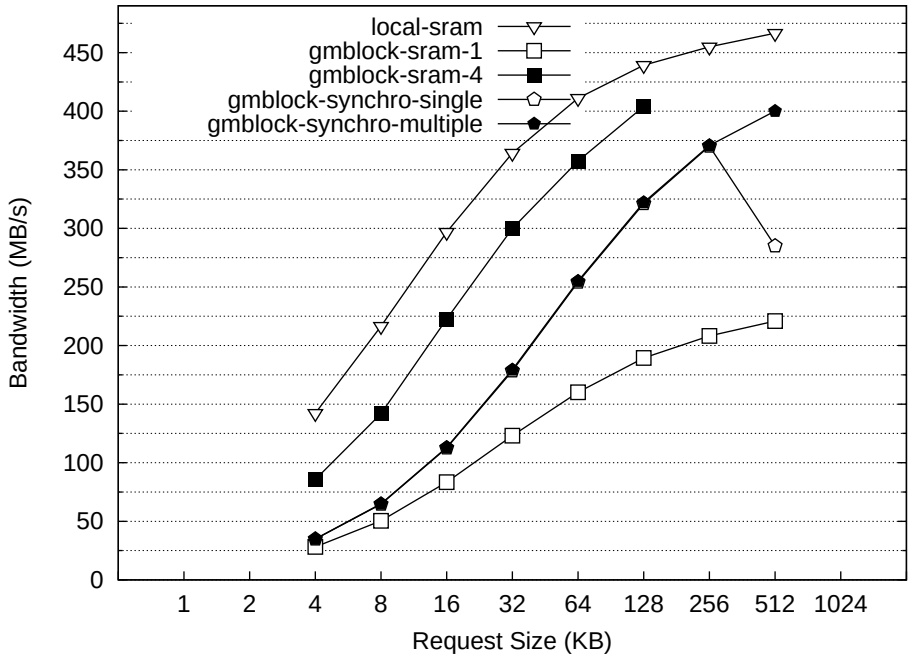
Το `gmblock-synchro-single` εμφανίζει σημαντικά βελτιωμένο ρυθμό μεταφοράς, 77% καλύτερο του `gmblock-sram` με χρήση της συσκευής MBL και αιτήσεις των 256KB. Υπάρχει μια αισθητή μείωση στην επίδοση του για αιτήσεις των 512KB στην οποία επικεντρωνόμαστε παρακάτω. Σε αντίθεση με τη συσκευή MBL, η βελτίωση για τη διάταξη RAID είναι μικρή, της τάξης του 7%.

Πείραμα 2β: Κίνηση δεδομένων από διατάξεις RAID

Για την κατανόηση της χαμηλής επίδοσης με χρήση διάταξης RAID και για αιτήσεις των 512KB με τη συσκευή MBL, χρειάζεται διερεύνηση του τρόπου με τον οποίο οι δοσοληψίες DMA από το αποθηκευτικό μέσο εξελίσσονται ως προς το χρόνο. Γράψαμε ένα εργαλείο ειδικού σκοπού, το `dma_poll`, το οποίο ιχνηλατεί την κίνηση των δεδομένων, με τεχνική παρόμοια της υλοποίησης συγχρονισμένων λειτουργιών αποστολής (Σχ. 23). Βρίσκουμε δύο λόγους για τη μειωμένη απόδοση:



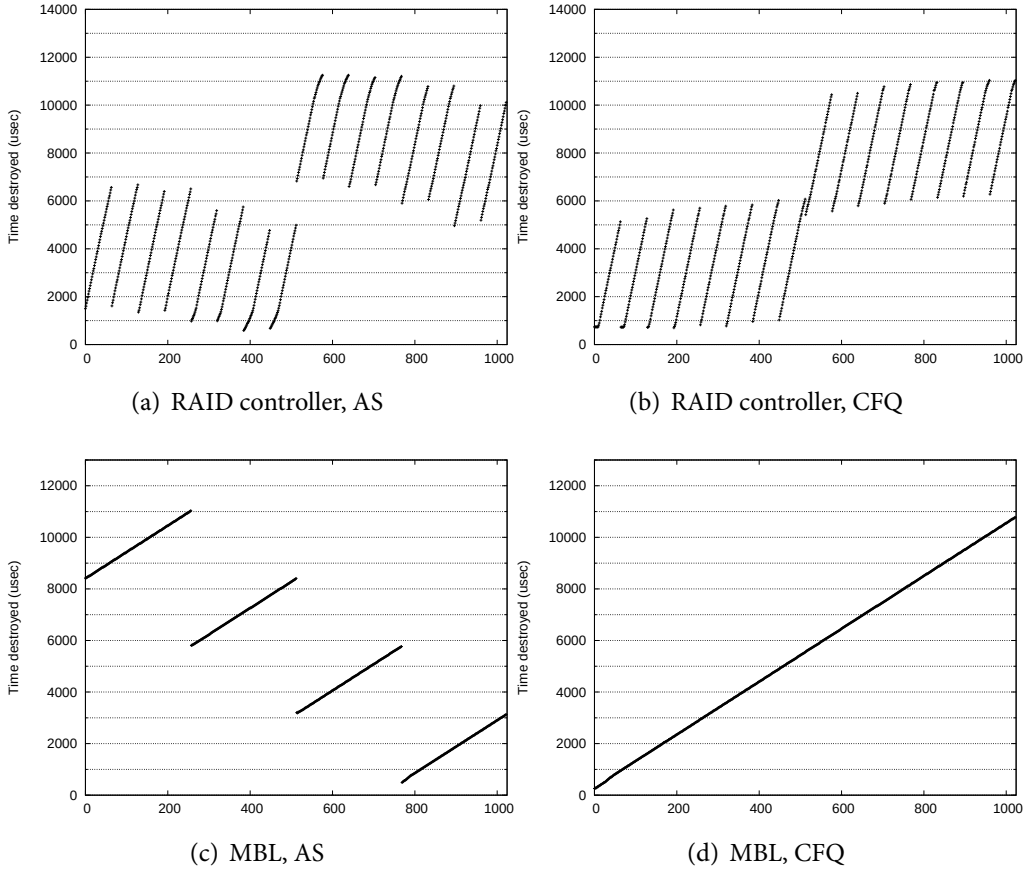
(a) Ρυθμός μεταφοράς, ελεγκτής RAID



(b) Ρυθμός μεταφοράς, MBL

Σχήμα 22: *Ρυθμός μεταφοράς απομακρυσμένων αναγνώσεων για συγχρονισμένες λειτουργίες GM*

Χρήση διάταξης RAID Το `gmblock-synchro-single` αγνοεί το γεγονός ότι τα δεδομένα τοποθετούνται παράλληλα σε διαφορετικά τμήματα της μνήμης (Σχ. 23(b)) και επικαλύπτει ανάγνωση και αποστολή μόνο για το πρώτο ρεύμα δεδομένων.



(a) RAID controller, AS (b) RAID controller, CFQ

(c) MBL, AS (d) MBL, CFQ

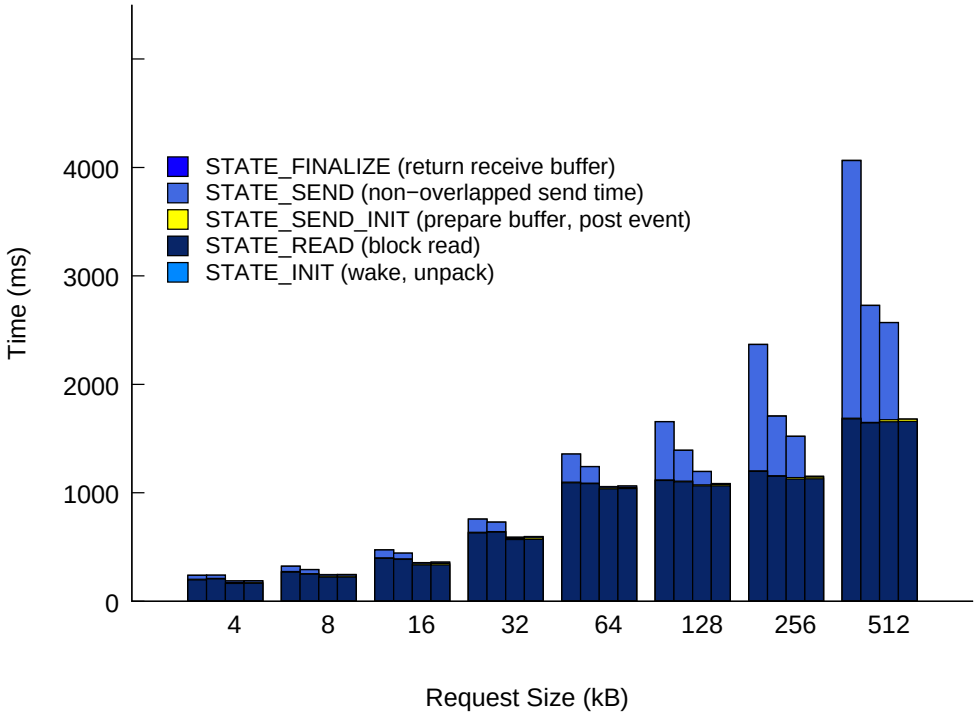
Σχήμα 23: *Ίχνη DMA για τους χρονοδρομολογητές E/E anticipatory και CFQ, αιτήσεις 1024KB*

Αλλαγή σειράς τμημάτων Το μέγιστο τμήμα που υποστηρίζει η συσκευή MBL για DMA είναι μήκους 256KB. Για αιτήσεις μεγαλύτερες αυτού του μεγέθους, η σειρά υποβολής των τμημάτων εξαρτάται από το χρονοδρομολογητή E/E του ΛΣ. Βρήκαμε ότι με χρήση του χρονοδρομολογητή AS (anticipatory) τα δύο τμήματα εναλλάσσονται (Σχ. 23(c)), οπότε ο βαθμός επικάλυψης είναι μικρότερος.

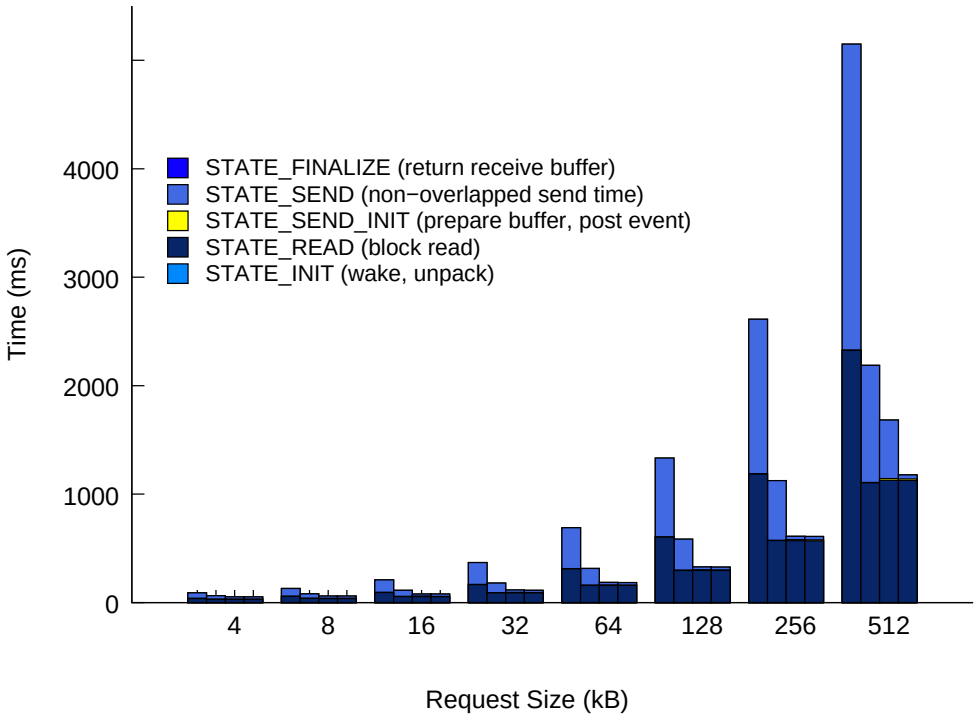
Το `gmblock-synchro-multiple` παρακάμπτει και τα δύο προβλήματα, επιτρέποντας την συλλογή δεδομένων για εξερχόμενα πακέτα από διαφορετικές περιοχές του απομονωτή, επιτυγχάνοντας ρυθμούς μεταφοράς πολύ κοντά στον τοπικό (92% του τοπικού), 40% καλύτερο από το `gmblock-sram-4`.

Τέλος, στο Σχ. 24 παρουσιάζεται η κατανομή σε διακριτές φάσεις του απαιτούμενου χρόνου εξυπηρέτησης ανά αίτηση.

Διακρίνουμε πέντε διαφορετικές φάσεις:



(a) RAID controller



(b) MBL

Σχήμα 24: *Διαίρεση του χρόνου αρχικής απόκρισης σε φάσεις, για τα gmblock-{ram, sram, synchro-single, synchro-multiple}*

- `STATE_INIT`: Λήψη και αποκωδικοποίηση της αίτησης.

- `STATE_READ`: Υποβολή αίτησης Ε/Ε προς το αποθηκευτικό μέσο και αναμονή για ολοκλήρωσή της.
- `STATE_SEND_INIT`: Υποβολή αίτησης δικτυακής Ε/Ε προς το Lanai.
- `STATE_SEND`: Η ανάγνωση έχει ολοκληρωθεί, η δικτυακή αποστολή είναι σε εξέλιξη.
- `STATE_FIN`: Επιβεβαίωση ολοκλήρωσης της δικτυακής αποστολής.

Ο χρόνος στις φάσεις πλην των `STATE_READ` και `STATE_SEND` ήταν αμελητέος. Η φάση `STATE_SEND` παριστά χρόνο δικτυακής Ε/Ε που δεν επικαλύπτεται με Ε/Ε από το δίσκο και υποδεικνύει το βαθμό επικάλυψης ανάγνωσης και αποστολής που επιτυγχάνεται. Στην περίπτωση του `gmblock-synchro-multiple` σχεδόν εξαλείφεται.

0.5 Σχεδίαση πελάτη και αποτίμηση από άκρο σε άκρο

Έως τώρα, εστιάσαμε κυρίως σε βελτιστοποίηση του μονοπατιού δεδομένων στην πλευρά του εξυπηρετητή. Το παρόν μέρος παρουσιάζει το σχεδιασμό και την υλοποίηση του gmblock στην πλευρά του πελάτη.

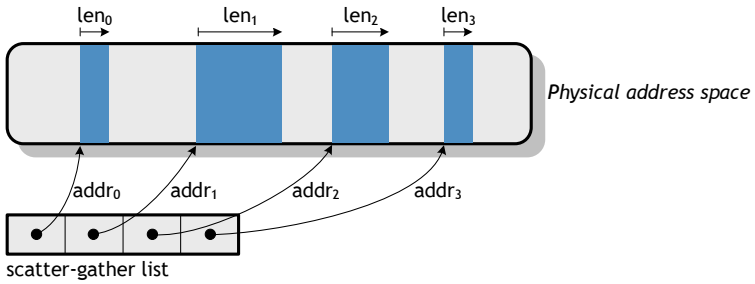
Ξεκινώντας από ένα βασικό πελάτη χώρου πυρήνα, εξετάζουμε την απόδοση εναλλακτικών σχεδιαστικών επιλογών κατά τη μετακίνηση δεδομένων. Για την αποδοτικότερη λειτουργία του πελάτη, εκμεταλλευόμαστε την προγραμματισιμότητα του Myrinet: προτείνουμε επεκτάσεις στο GM ώστε να υποστηρίζει Ε/Ε με διασπορά και συλλογή δεδομένων (scatter-gather I/O) απευθείας από και προς το χώρο φυσικών διευθύνσεων και εισάγουμε τη χρήση τους στον πελάτη του gmblock. Τέλος, εγκαθιστούμε ένα παράλληλο σύστημα αρχείων πάνω από το gmblock, το οποίο επιτρέπει την αποτίμηση της επίδοσης του συστήματος από άκρο σε άκρο με ρεαλιστικά μετροπρογράμματα.

0.5.1 Σχεδίαση ενός πελάτη nbd επιπέδου χρήστη

Κίνηση δεδομένων στην πλευρά του πελάτη

Στο Σχ. 8(a) παρουσιάζεται η βασική σχεδίαση ενός συστήματος nbd. Ο πελάτης nbd χρειάζεται να συλλέξει δεδομένα μπλοκ από απομονωτές Ε/Ε μέσα σε δικτυακά μηνύ-

ματα (εντολές εγγραφής) και να διασπείρει δεδομένα από δικτυακά μηνύματα σε απομονωτές Ε/Ε (εντολές ανάγνωσης). Σε κάθε περίπτωση, οι απομονωτές περιγράφονται στη βάση μιας λίστας διασποράς-συλλογής δεδομένων που περιγράφει τη θέση των δεδομένων στη μνήμη (Σχ. 25). Η επιβάρυνση που εισάγεται στο σημείο αυτό εξαρτάται από τις δυνατότητες της υφιστάμενης δικτυακής υποδομής. Ακόμη και όταν χρησιμοποιείται ένα δίκτυο επικοινωνίας χώρου χρήστη, η μετάφραση από αιτήσεις Ε/Ε που αναφέρονται σε διακριτές περιοχές φυσικής μνήμης σε μηνύματα του GM δεν είναι απλή. Υπάρχει ασυμφωνία ανάμεσα στις ανάγκες ενός πελάτη nbd χώρου πυρήνα και τις δυνατότητες ενός συστήματος όπως το GM.

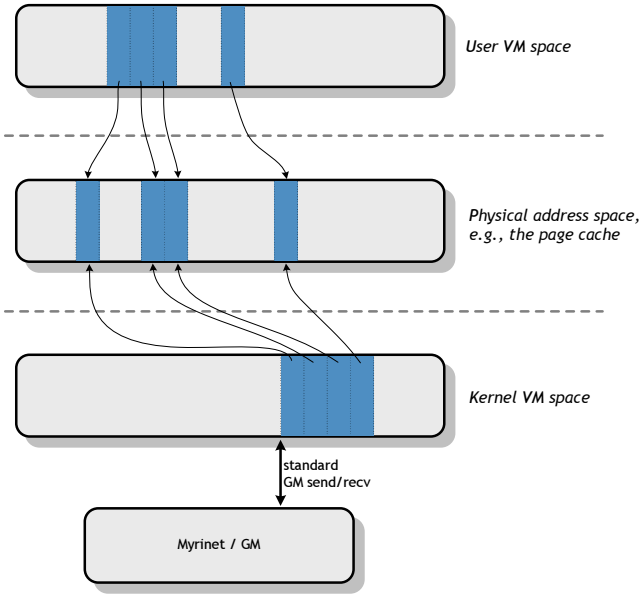


Σχήμα 25: *Μία λίστα διασποράς-συλλογής περιγράφει έναν απομονωτή Ε/Ε διάσπαρτο στη φυσική μνήμη*

Το GM απαιτεί η επικοινωνία να γίνεται από απομονωτές συνεχόμενους στο χώρο εικονικής μνήμης των διεργασιών, με πρότερη δήλωση των ανάλογων περιοχών. Διάσπαρτες περιοχές φυσικής μνήμης υποστηρίζονται μέσω της διαδικασίας μετάφρασης εικονικών διευθύνσεων σε φυσικές.

Από την άλλη πλευρά, ο πελάτης nbd χώρου χρήστη δέχεται αιτήσεις που αφορούν τμήματα *φυσικής* μνήμης. Διάφορες προσεγγίσεις μπορούν να χρησιμοποιηθούν για την αντιμετώπιση του προβλήματος:

Ενδιάμεση αποθήκευση σε προ-δηλωμένους απομονωτές: Ο πελάτης nbd δεσμεύει και δηλώνει στο GM έναν αριθμό απομονωτών, κατά την αρχικοποίησή του. Όλα τα δεδομένα μπλοκ αποθηκεύονται ενδιάμεσα σε αυτούς· η CPU *συλλέγει* δεδομένα πριν από αίτηση εγγραφής, ή *διασπείρει* εισερχόμενα δεδομένα μετά από αίτηση ανάγνωσης. Παρά την απλότητά της, η σχεδίαση αυτή εμφανίζει σημαντικά μειονεκτήματα:*(a)* εισάγει σημαντική επιβάρυνση στη CPU του κόμβου λόγω αντιγραφών, *(b)* καταναλώνει το τριπλάσιο εύρος ζώνης στο διάδρομο μνήμης σε σχέση με το ρυθμό απομακρυσμένης Ε/Ε που επιτυγχάνεται, *(c)* χρειά-

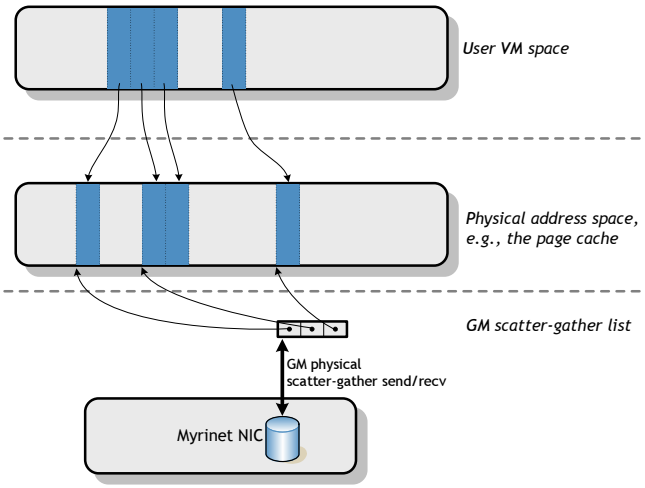


Σχήμα 26: *Απεικόνιση απομονωτών σε συνεχόμενες εικονικές διευθύνσεις του πυρήνα*

ζεται μεταβολή στους πίνακες σελίδων του πυρήνα μέσα στο κρίσιμο μονοπάτι αφού χρησιμοποιείται προγραμματιζόμενη Ε/Ε, με απενεργοποιημένες τις διακοπές και κλειδωμένη την εικονική συσκευή Ε/Ε.

Απεικόνιση διάσπαρτων φυσικών σελίδων σε συνεχόμενη εικονική μνήμη του πυρήνα: Ο πελάτης nbd συναρμολογεί διάσπαρτες σελίδες φυσικής μνήμης σε μια συνεχόμενη περιοχή εικονικών διευθύνσεων, την οποία δηλώνει στο GM και τη χρησιμοποιεί για λειτουργίες δικτυακής Ε/Ε. Παρόλο που η σχεδίαση είναι χωρίς αντίγραφα – ο προσαρμογέας δικτύου διασχίζει γραμμικά το χώρο εικονικής μνήμης και κάνει DMA σε ασύνδετες σελίδες φυσικής μνήμης – εισάγει δηλώσεις περιοχών και ακυρώσεις μέσα στο κρίσιμο μονοπάτι. Επιπλέον, τηρεί αντιστοιχίες μνήμης προς τις αναφερόμενες φυσικές σελίδες και εμπλέκει την κάρτα στη διαδικασία μετάφρασης, ενώ αυτό είναι απαραίτητο μόνο κατά την επικοινωνία χώρου χρήστη. Στην υπό μελέτη περίπτωση ο πυρήνας είναι *ήδη* στο κρίσιμο μονοπάτι. Μάλιστα, ο καλών, το στρώμα συσκευών μπλοκ, έχει φροντίσει να καθηλώσει (pin down) όσες σελίδες χρειάζεται, όταν π.χ. γίνεται Ε/Ε τύπου `O_DIRECT`, οπότε παρέχει απευθείας μια λίστα διασποράς-συλλογής δεδομένων με φυσικά τμήματα.

Χρήση μονόπλευρων λειτουργιών RDMA στην πλευρά του εξυπηρετητή: Ο εξυπηρετητής χρησιμοποιεί μονόπλευρες λειτουργίες απομακρυσμένης ανάγνωσης και



Σχήμα 27: *Υποστήριξη του GM για Ε/Ε με τμήματα φυσικής μνήμης*

εγγραφής δεδομένων (RDMA read/write) ώστε να τοποθετεί δεδομένα απευθείας στη μνήμη των πελατών nbd. Με αυτόν τον τρόπο, είτε ο εξυπηρετητής έχει απεριόριστη πρόσβαση στο χώρο φυσικών διευθύνσεων, είτε παραμένει η ανάγκη για δήλωση περιοχών μνήμης από τους πελάτες στο κρίσιμο μονοπάτι.

Καθώς οι παραπάνω προσεγγίσεις εισάγουν επιβάρυνση στη λειτουργία ενός πελάτη nbd χώρου πυρήνα, προκύπτει η ανάγκη για συνδυασμό της ασφαλούς, αμφίπλευρης επικοινωνίας του GM με μη συνεχόμενους απομονωτές στη φυσική μνήμη, όπως προσδιορίζονται από το υπερκείμενο στρώμα συσκευών μπλοκ του ΛΣ. Για το σκοπό αυτό προτείνουμε επεκτάσεις στο μηχανισμό του GM ώστε να υποστηρίζει Ε/Ε με συλλογή και διασπορά δεδομένων απευθείας από απομονωτές ορισμένους με λίστες φυσικών διευθύνσεων (Σχ. 27).

Ορίζουμε μια νέα κλάση λειτουργιών αποστολής και λήψης και αντίστοιχες αιτήσεις για το χειρισμό τους. Ο σχετικός απομονωτής καθορίζεται ως μια λίστα διασποράς-συλλογής φυσικών διευθύνσεων. Η κατάσταση κάθε θύρας του GM επεκτείνεται ώστε να υποστηρίζει έναν αριθμό λιστών διασποράς-συλλογής, κάθε μία από τις οποίες περιέχει έναν αριθμό φυσικών τμημάτων, δηλ. ζευγών {φυσική διεύθυνση τμήματος, μήκος τμήματος }.

Για την πραγματοποίηση μιας λειτουργίας Ε/Ε, ο καλών:

1. αρχικοποιεί μια λίστα διασποράς-συλλογής (`gm_set_scatterlist()`),

2. υποβάλλει μια αίτηση αποστολής (`gm_gather_send_with_callback()`) ή λή-

ψης (`gm_provide_scatter_receive_buffer_with_tag()`) δεδομένων με αναφορά στη λίστα,

3. η οποία ανήκει στο Lanai, έως ότου...
4. ο χρήστης ενημερωθεί για την ολοκλήρωση της διαδικασίας μέσω της ουράς συμβάντων.

Η σχεδίαση διατηρεί τη σημασιολογία ταιριάσματος μηνυμάτων του GM, βάσει του χαρακτηριστικού *size* που διαθέτει κάθε αίτηση επικοινωνίας με λίστα διασποράς-συλλογής. Έτσι, το ταίριασμα γίνεται εξ ολοκλήρου στην κάρτα, η οποία τοποθετεί τα δεδομένα κάθε πακέτου απευθείας στην τελική θέση τους, χωρίς δηλώσεις περιοχών μνήμης. Η CPU διακόπτεται μόνο όταν η λειτουργία επικοινωνίας ολοκληρωθεί στο σύνολό της.

Υλοποίηση λειτουργιών διασποράς-συλλογής στο GM

Η υλοποίηση λειτουργιών διασποράς-συλλογής στο GM επηρεάζει και τα τρία τμήματά του. Η βιβλιοθήκη χώρου χρήστη εμπλουτίζεται με τις υπηρεσίες `gm_set_scatterlist()`, `gm_gather_send_with_callback()` και `gm_provide_scatter_receive_buffer_with_tag()`. Ο οδηγός συσκευής του GM επεκτείνεται ώστε να υποστηρίζει χωριστές λίστες διασποράς-συλλογής ανά θύρα του GM. Το πλήθος τους και ο αριθμός των τμημάτων που μπορεί να περιέχει κάθε μία είναι σταθερές χρόνου μεταγλώττισης. Αυξημένος αριθμός τμημάτων ευνοεί τη διαχείριση μεγαλύτερων αιτήσεων προς τον εξυπηρετητή nbd, αλλά δεσμεύει μεγαλύτερο χώρο στη μνήμη του Lanai για την αποθήκευση των λιστών. Τέλος, το μεγαλύτερο τμήμα της υποστήριξης λειτουργικών διασποράς-συλλογής υλοποιείται στο υλικολογισμικό. Η μηχανή SDMA μεταβάλλεται ώστε κατά τη διάσχιση μιας λίστας συλλογής να τηρεί την τρέχουσα κατάσταση του μηνύματος ως την τριάδα {τρέχουσα φυσική διεύθυνση, τρέχον τμήμα στη λίστα συλλογής, πλήθος bytes που απομένουν}. Κάθε πακέτο δημιουργείται άμεσα από τη φυσική διεύθυνση χωρίς μετάφραση. Όταν ένα τμήμα της λίστας εξαντληθεί, ο δείκτης φυσικής διεύθυνσης αρχικοποιείται με βάση το επόμενο. Ομοίως επεκτείνεται και ο μηχανισμός επαναποστολής πακέτων. Στην πλευρά του παραλήπτη, η μηχανή RDMA αναλαμβάνει να διατρέχει για κάθε εισερχόμενο πακέτο τη λίστα διασποράς, ώστε να καθορίσει την τελική διεύθυνση DMA. Η λίστα δεν διατρέχεται σειριακά, καθώς πρέπει να υποστηρίζεται ο συνδυασμός λήψης με λίστα διασποράς και αποστολής με συγχρονισμό, οπότε

τα εισερχόμενα τεμάχια δεδομένων μπορεί να καταλήξουν σε οποιοδήποτε σημείο του απομονωτή. Η τελική διεύθυνση DMA υπολογίζεται με βάση την τιμή του πεδίου κεφαλίδας `h_synchro_ptr` του πακέτου και την αρχική φυσική διεύθυνση του αντίστοιχου τμήματος της λίστας διασποράς.

0.5.2 Πελάτης gmblock χώρου πυρήνα

Στην πλευρά του πελάτη το gmblock εκτελείται ως οδηγός συσκευής χώρου πυρήνα, υλοποιώντας την καθιερωμένη διεπαφή συσκευών μπλοκ του ΛΣ. Με τον τρόπο αυτό το προτεινόμενο σύστημα nbd μπορεί να χρησιμοποιηθεί άμεσα είτε απευθείας ως συσκευή μπλοκ, π.χ. από ένα παράλληλο ΣΔΒΔ, είτε σε συνδυασμό με ένα παράλληλο σύστημα αρχείων μοιραζόμενου δίσκου.

Ο πελάτης χρησιμοποιεί τις επεκτάσεις του GM που περιγράφηκαν στα προηγούμενα. Κάθε αίτηση Ε/Ε του ΛΣ απεικονίζεται για πραγματοποίηση DMA και η λίστα διασποράς-συλλογής που προκύπτει προωθείται ως έχει στο GM για εξυπηρέτηση. Τα όρια της ουράς αιτήσεων του οδηγού τίθενται με βάση τις παραμέτρους χρόνου μεταγλώττισης του GM: το μήκος της ουράς καθορίζεται από το πλήθος των λιστών του GM και ο αριθμός των τμημάτων ανά λίστα καθορίζει το μέγιστο αριθμό ασύνδετων περιοχών φυσικής μνήμης ανά αίτηση.

Στην περίπτωση αιτήσεων απομακρυσμένης ανάγνωσης, το ταίριασμα των εισερχόμενων πακέτων γίνεται εξ ολοκλήρου στην κάρτα δικτύου, χωρίς παρέμβαση της CPU. Ο οδηγός χρησιμοποιεί διακριτό όρισμα *size* για κάθε αίτηση του GM και χωριστή λίστα διασποράς για να το επιτύχει αυτό. (Σχ. 27). Ο συνδυασμός αυτής της σχεδίασης με το προτεινόμενο μονοπάτι δεδομένων στην πλευρά του εξυπηρετητή επιτρέπει απευθείας μετακίνηση δεδομένων από το αποθηκευτικό μέσο στο δίκτυο και σε τελικούς απομονωτές χώρου χρήστη, με μηδενικά αντίγραφα. Εξ όσων γνωρίζουμε, αυτή είναι η πρώτη σχεδίαση και υλοποίηση που υποστηρίζει τέτοια λειτουργικότητα.

0.5.3 Αποτίμηση παράλληλου συστήματος αρχείων

Εγκαταστήσαμε το παράλληλο σύστημα αρχείων Oracle Cluster File System (OCFS2) πάνω από μοιραζόμενο χώρο αποθήκευσης υλοποιημένο με χρήση του gmblock, ώστε

να αποτιμήσουμε την επίδοσή του με ρεαλιστικά μοτίβα πρόσβασης διαφόρων φορτίων.

Η πειραματική εγκατάσταση αποτελείται από τέσσερις κόμβους τύπου A και ένα αποθηκευτικό εξυπηρετητή τύπου B. Όλοι οι κόμβοι μοιράζονται πρόσβαση σε διάταξη RAID0 μέσω του πελάτη gmblock χώρου χρήστη. Τρέχουν την έκδοση OCFS2 1.5.0, η οποία αποτελεί μέρος του πυρήνα Linux 2.6.28.2.

Η χρήση του προτεινόμενου μονοπατιού δεν επιτρέπει την προανάκτηση δεδομένων και την προσωρινή αποθήκευσή του στην πλευρά του εξυπηρετητή. Μελετούμε την επίδρασή τους συγκρίνοντας δύο εκδόσεις του gmblock: την `gmblock-ramcache`, η οποία περνά κάθε αίτηση E/E από την κρυφή μνήμη σελίδων του πυρήνα και την `gmblock-sram`, η οποία χρησιμοποιεί άμεση E/E και κινεί δεδομένα μέσω του προτεινόμενου μονοπατιού. Οι εγγραφές συνεχίζουν να περνούν μέσω της κύριας μνήμης χωρίς προσωρινή αποθήκευση, για να παρακάμψουμε τον περιορισμό του Lanai που περιγράφεται στο § 0.3.3.

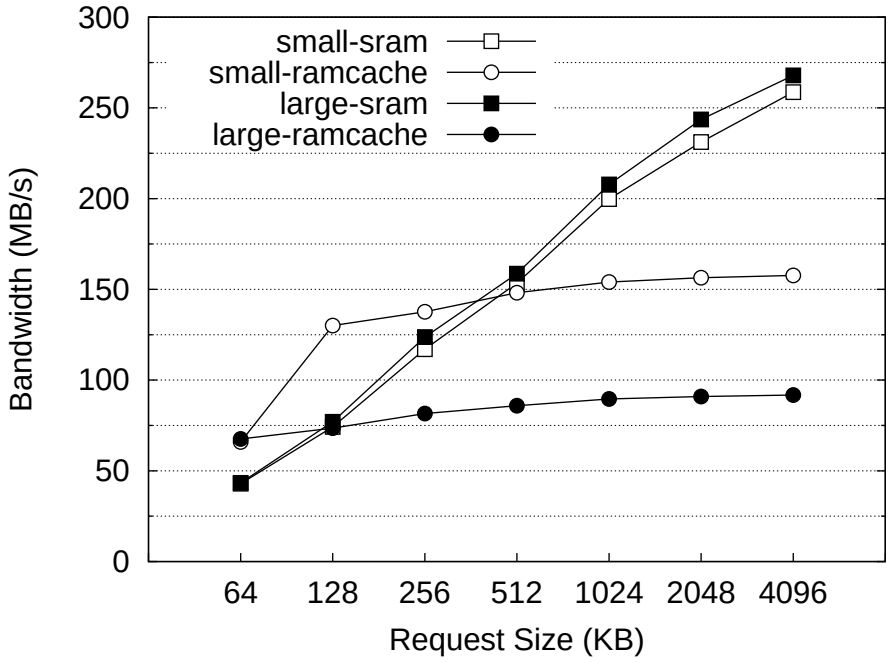
Τρέχουμε τρία διαφορετικά μετροπρογράμματα: το IOzone [NC], ένα φορτίο εξυπηρετητή ιστοσελίδων και το παράλληλο πρόγραμμα MPI-Tile-I/O [Ros].

Πείραμα 3α: Επίδοση του IOzone με έναν κόμβο

Το IOzone είναι ένα μετροπρόγραμμα για συστήματα αρχείων που παράγει διάφορα μοτίβα πρόσβασης. Το εκτελούμε σε κατάσταση ανάγνωσης, επανάγνωσης και εγγραφής. Σε κάθε δοκιμή, το IOzone εκτελεί πολλαπλά περάσματα, μεταβάλλοντας το μέγεθος της αίτησης E/E από 64KB έως 4096KB. Χρησιμοποιούμε δύο διαφορετικά φορτία: ένα «μικρό» αρχείο μήκους 512MB το οποίο χωρά ολόκληρο στην κρυφή μνήμη ενός από τους κόμβους κι ένα «μεγάλο» αρχείο, μήκους 4GB, ώστε να γίνει εμφανής η επίδραση της κρυφής μνήμης στον εξυπηρετητή και τους πελάτες.

Στο Σχ. 28 φαίνεται η επίδοση με χρήση ενός πελάτη, με απευθείας E/E, βασισμένη στο μηχανισμό `O_DIRECT`. Το `gmblock-ramcache` περιορίζεται από το εύρος ζώνης του συνδέσμου μνήμης-περιφερειακού δρόμου στα ∼93MB/s and ∼160MB/s αντίστοιχα. Η επίδοση για το μικρό αρχείο είναι αισθητά καλύτερη γιατί το πρώτο πέρασμα των 64KB το φέρνει ολόκληρο στην κρυφή μνήμη του εξυπηρετητή.

Το `gmblock-sram` κλιμακώνεται γραμμικά με την αύξηση του μεγέθους της αίτησης,

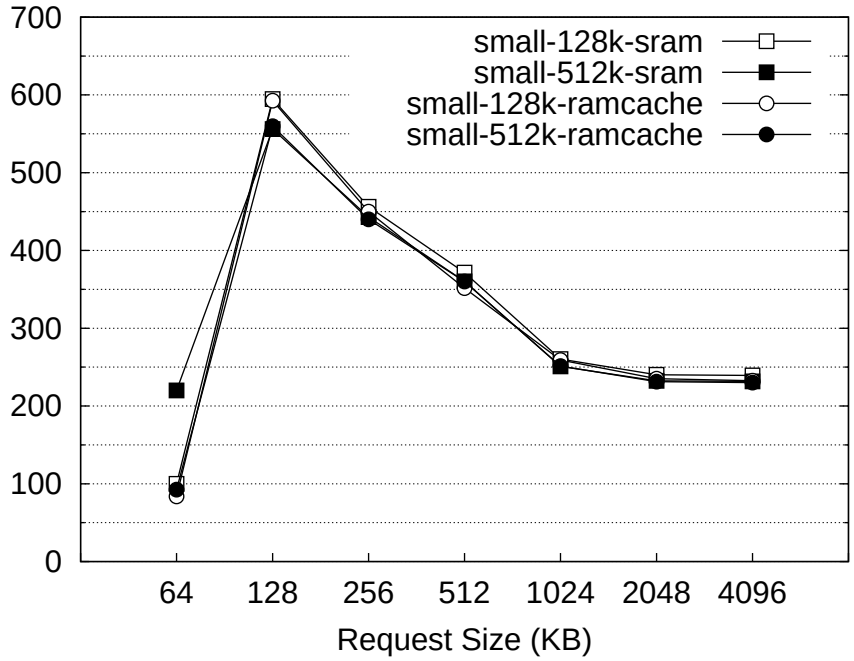


Σχήμα 28: *IOzone, ένας πελάτης: χωρίς κρυφή μνήμη, άμεση Ε/Ε*

εφόσον δεν γίνεται προσωρινή αποθήκευση ούτε στον πελάτη ούτε στον εξυπηρετητή. Επιπλέον, το `gmblock-ram` είναι καλύτερο του `gmblock-sram` για το αρχικό πέρασμα των 64KB, διότι ο εξυπηρετητής μπορεί να προανακτήσει δεδομένα στην κρυφή του μνήμη, εκτελώντας αιτήσεις των 512KB. Το `gmblock-sram` ξεπερνά το `gmblock-ramcache` μετά από αυτό το όριο και είναι ∼1.64 και ∼2.9 φορές καλύτερο για το μικρό και το μεγάλο αρχείο αντίστοιχα.

Στο Σχ. 29 φαίνεται η επίδοση για το μικρό αρχείο, με ενεργοποιημένη την κρυφή μνήμη στους πελάτες και μήκος προανάκτησης 128KB και 512KB. Βλέπουμε την επίδραση της προσωρινής αποθήκευσης στην πλευρά των πελατών, καθώς κάθε ένας αποθηκεύει χωριστά το αρχείο στην κρυφή μνήμη σελίδων του. Η απότομη μείωση καθώς αυξάνεται το μέγεθος της αίτησης οφείλεται στο μέγεθος της κρυφής μνήμης L2 του επεξεργαστή: καθώς το μέγεθος της αίτησης αυξάνεται, ο απομονωτής χώρου χρήστη του IOzone αποθηκεύεται εκτός της κρυφής μνήμης L2. Βλέπουμε επίσης ότι η προανάκτηση δεδομένων είναι απαραίτητη για την επίτευξη ικανοποιητικής απόδοσης. Με χρήση του προτεινόμενου μονοπατιού δεν είναι δυνατή η προανάκτηση στον εξυπηρετητή, μπορεί όμως να προκληθεί στην πλευρά του πελάτη. Στα υπόλοιπα, συνεχίζουμε με μήκος προανάκτησης 512KB στους πελάτες.

Τέλος, το Σχ. 30 παρουσιάζει την επίδοση για το μεγάλο αρχείο, όπου ο βαθμός χρησι-

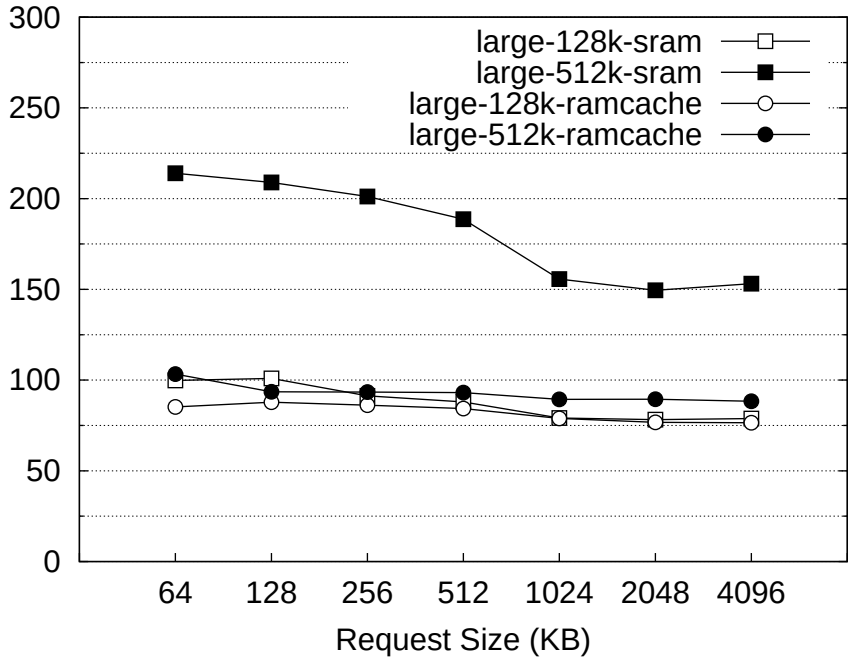


Σχήμα 29: *IOzone, ένας πελάτης: κρυφή μνήμη στην πλευρά του πελάτη, μικρό αρχείο*

μοποίησης της κρυφής μνήμης είναι μικρός.

Πείραμα 3β: Επίδοση IOzone με πολλούς κόμβους

Επαναλαμβάνουμε τα προηγούμενα πειράματα, αυτή τη φορά με τέσσερα στιγμιότυπα του IOzone να εκτελούνται παράλληλα, ένα σε κάθε κόμβο-πελάτη. Ο συνολικός ρυθμός μεταφοράς που επιτυγχάνεται για άμεση Ε/Ε παρουσιάζεται στο Σχ. 31. Η επεξεργασία του ίδιου αρχείου των 512MB ή των 4GB είναι η βέλτιστη περίπτωση για το `gmblock-ramcache`: εμφανίζει σταθερά καλή επίδοση, καθώς όλοι οι κόμβοι επωφελούνται ταυτόχρονα από κάθε μεταφορά από το δίσκο στην κρυφή μνήμη του εξυπηρετητή, καθώς κινούνται με περίπου τον ίδιο ρυθμό μέσα στο αρχείο. Από την άλλη πλευρά, ήταν δυσκολότερο για το `gmblock-sram` να έχει σταθερή απόδοση. Έγινε γρήγορα φανερό ότι η επιλογή του χρονοδρομολογητή Ε/Ε στην πλευρά του εξυπηρετητή ήταν πολύ σημαντική για την επίτευξη καλής επίδοσης. Δοκιμάσαμε με τους τέσσερις διαφορετικούς χρονοδρομολογητές που παρέχει ο πυρήνας του Linux, (AS, deadline, noop, CFQ) και παρουσιάζουμε για λόγους απλότητας αποτελέσματα από τους AS και deadline. Η επίδοση μεταβάλλεται σημαντικά με το μέγεθος της αίτησης· στην καλύτερη περίπτωση η απόδοση των δίσκων ήταν αρκετά υψηλή ώστε το `gmblock-sram` να ξεπεράσει το `gmblock-ramcache` κατά 66%. Στη χειρότερη περίπτωση, το `gmblock-`



Σχήμα 30: *IOzone, ένας πελάτης: κρυφή μνήμη στην πλευρά του πελάτη, μεγάλο αρχείο*

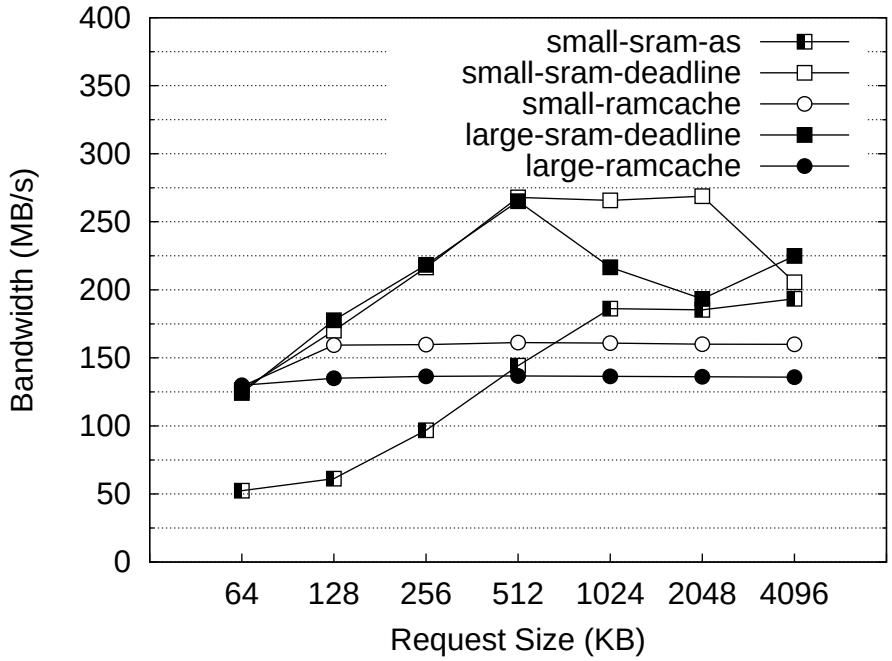
`sram` επιτυγχάνει μόνο το 40% της επίδοσης του `gmblock-ramcache`, με αιτήσεις των 64KB και χρήση του anticipatory scheduler.

Στο Σχ. 32 φαίνεται η επίδοση για το μικρό αρχείο. Μετά την αρχική ανάγνωσή του, όλοι πελάτες ανακτούν δεδομένα παράλληλα από τις κρυφές μνήμες σελίδων τους, στα ∼2.4GB/s.

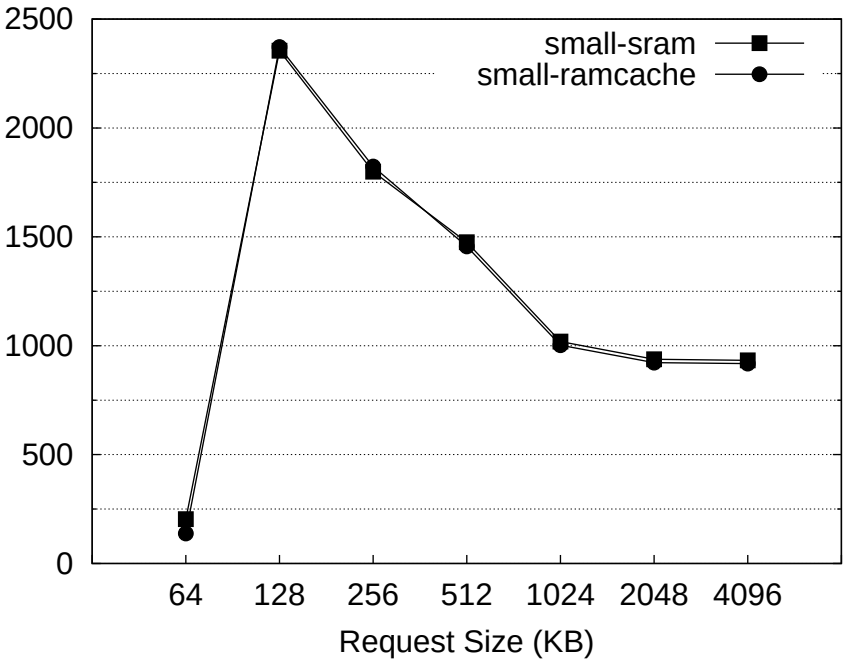
Στο Σχ. 33 φαίνεται η επίδοση για μεγάλο αρχείο και 512KB μήκος προανάκτησης. Το `gmblock-sram` είναι σταθερά καλύτερο του `gmblock-ramcache`, αλλά η επίδοσή του είναι ευαίσθητη στο μέγεθος της αίτησης, κάτι που αποδίδουμε στην αλληλεπίδραση του χρονισμού των αιτήσεων με το μηχανισμό χρονοδρομολόγησης E/E στην πλευρά του εξυπηρετητή.

Πείραμα 3γ: Φορτίο εξυπηρετητή

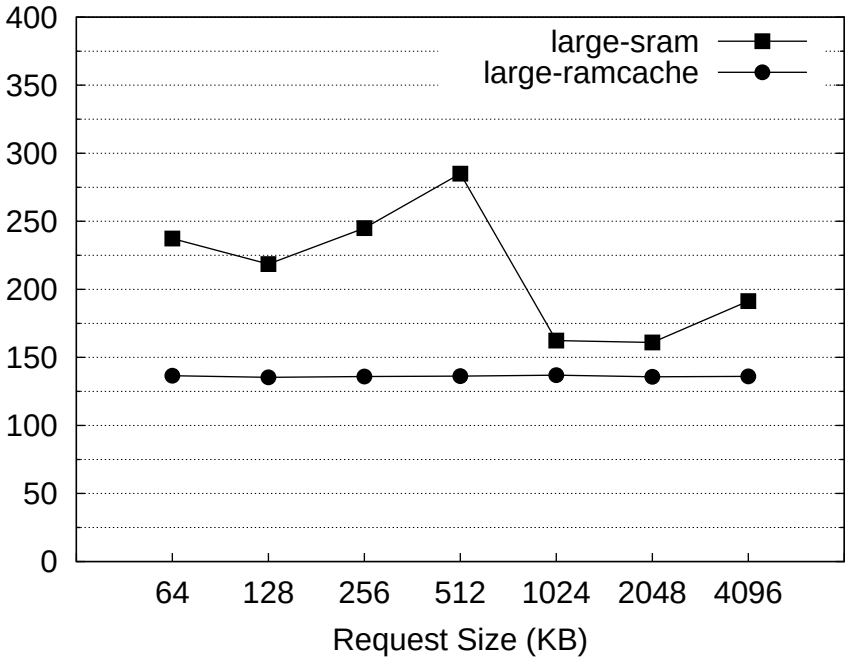
Αποτιμούμε την επίδοση της εγκατάστασης του OCFS2 σε φορτία εξυπηρετητή. Προσομοιώνουμε ένα σενάριο όπου τέσσερις εξυπηρετητές Ιστού διαβάζουν αρχεία από το κοινό σύστημα αρχείων για να ολοκληρώσουν την επεξεργασία εισερχόμενων αιτήσεων. Κάθε αίτηση αφορά τυχαία επιλεγμένο αρχείο και ο αριθμός των αιτήσεων που ολοκληρώνονται σε διάστημα δύο λεπτών χρησιμοποιείται ως μετρική επίδοσης του



Σχήμα 31: *IOzone, πολλοί πελάτες: χωρίς κρυφή μνήμη, άμεση Ε/Ε*



Σχήμα 32: *IOzone, πολλοί πελάτες: κρυφή μνήμη στην πλευρά του πελάτη, μικρό αρχείο*



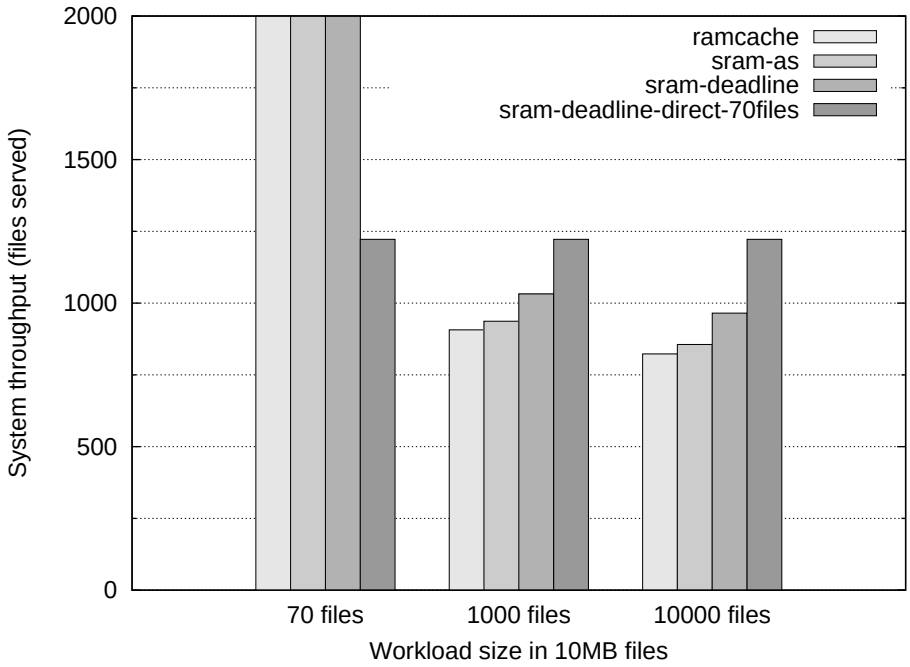
Σχήμα 33: *IOzone, πολλοί πελάτες: κρυφή μνήμη στην πλευρά του πελάτη, μεγάλο αρχείο*

συστήματος. Το μέγεθος του συνόλου αρχείων ποικίλει από ένα μικρό φορτίο που χωρά στην κρυφή μνήμη (70 αρχεία των 10MB το καθένα), έως 1000 και 10000 αρχεία των 10MB το καθένα (Σχ. 34(a)).

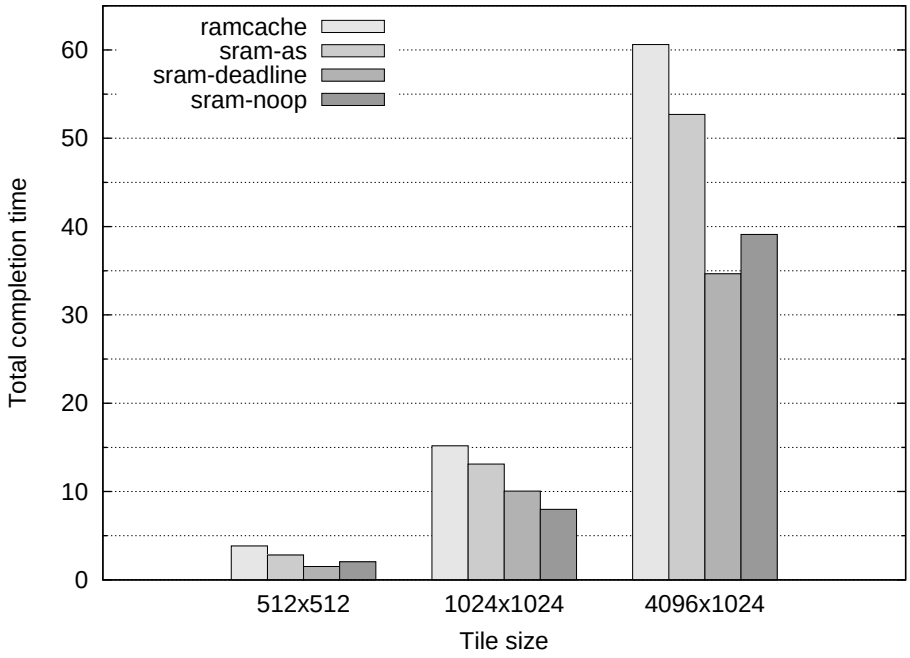
Για το φορτίο που χωρά στην κρυφή μνήμη, τα `gmblock-ramcache` και `gmblock-sram` δεν είχαν σημαντική διαφορά στην επίδοση (πέτυχαν ρυθμό ∼7000 αρχεία/λεπτό, είναι εκτός διαγράμματος). Όλοι οι πελάτες έφεραν το σύνολο των αρχείων στις τοπικές κρυφές μνήμες τους. Όταν ο αριθμός των αρχείων αυξήθηκε σε 1000 και 10000 αρχεία, το `gmblock-sram` ήταν κατά 12% και 17% καλύτερο αντίστοιχα. Η επίδοσή του περιορίζεται από την απόδοση των δίσκων. Για να επιβεβαιωθεί δοκιμάσαμε με ένα μικρό σύνολο αρχείων (70 αρχεία) χωρίς κρυφή μνήμη στους πελάτες (πρόσβαση `O_DIRECT`). Καθώς η αναζήτηση δεδομένων αφορούσε μικρότερη περιοχή του δίσκου, η απόδοση αυξήθηκε επιπλέον κατά 18% και 26% σε σύγκριση με την περίπτωση των 1000 και 10000 αρχείων αντίστοιχα.

Πείραμα 3δ: Εφαρμογή MPI-IO

Χρησιμοποιούμε το MPI-Tile-IO, ένα μετροπρόγραμμα για τη βιβλιοθήκη MPI-IO που εκτελεί μη συνεχόμενες προσβάσεις στο δίσκο με σταθερό βήμα, μοτίβο παρόμοιο με



(a) Φορτίο Εξυπηρετητή Ιστού



(b) Εφαρμογή MPI-Tile-I/O

Σχήμα 34: *Επίδοση για το φορτία εξυπηρετητή Ιστού και MPI-Tile-I/O*

αυτό που προκαλούν εφαρμογές οπτικοποίησης αποτελεσμάτων και αριθμητικές εφαρμογές. Το αρχείο εισόδου της εφαρμογής χωρίζεται σε ένα διδιάστατο σύνολο από υποπίνακες, και κάθε μία από τις διεργασίες MPI εκτελεί προσπελάσεις σε έναν από αυτούς.

Το πρόγραμμα προσομοιώνει την απεικόνιση μιας εικόνας αποτελούμενης από 4×4 περιοχές με μέγεθος περιοχής 512×512, 1024×1024 και 4096×4096 στοιχεία και 32 bytes ανά στοιχείο. Στο Σχ. 34(b) παρουσιάζεται ο συνολικός χρόνος εκτέλεσης του προγράμματος. Στη βέλτιστη περίπτωση το `gmblock-sram` εκτελείται σε χρόνο 39%, 51% και 57% του απαιτούμενου χρόνου για το `gmblock-ramcache`, αν και δεν υπάρχει επιλογή χρονοδρομολογητή Ε/Ε που να οδηγεί πάντα σε καλύτερη επίδοση.

Συνολικά, το `gmblock-sram` είχε καλύτερη επίδοση και στα τρία φορτία, ωστόσο η βελτίωση από τη χρήση του συντομότερου μονοπατιού δεδομένων γίνεται φανερή μόνο όταν η χρονοδρομολόγηση Ε/Ε στην πλευρά του εξυπηρετητή μπορέσει να άρει τη στενωπό επίδοσης στους δίσκους λόγω ταυτόχρονης πρόσβασης σε διαφορετικά σημεία του κοινού αποθηκευτικού μέσου.

0.6 Σχετικές εργασίες

Η παρούσα διατριβή περιγράφει ένα πλαίσιο για Ε/Ε επιπέδου μπλοκ πάνω από ευφυή δίκτυα διασύνδεσης με απευθείας μεταφορές ανάμεσα σε αποθηκευτικά μέσα και το δίκτυο.

Έτσι είναι σχετική με εργασίες από το χώρο των συστημάτων επικοινωνίας χώρου χρήστη, την αποδοτική χρήση μοιραζόμενων αρχιτεκτονικών πόρων σε συστοιχίες από SMPs, συστήματα μοιραζόμενης πρόσβασης επιπέδου μπλοκ, παράλληλα και κατανεμημένα συστήματα αρχείων και προγραμματιζόμενες δικτυακές αρχιτεκτονικές.

Στο § 0.2.2 γίνεται σύντομη αναφορά σε συστήματα επικοινωνίας χώρου χρήστη και το λογισμικό Myrinet/GM που χρησιμοποιείται ως η δικτυακή υποδομή του gmblock.

Πολλές εργασίες επικεντρώνονται στο πρόβλημα του περιορισμένου εύρους ζώνης διαδρόμων σε συστήματα SMP [LVE00, Bel97, Sch03], εστιάζοντας στην επίπτωση που έχει ο ανταγωνισμός για πρόσβαση στη μνήμη στο συνολικό χρόνο εκτέλεσης εφαρμογών, ενώ έχουν προταθεί τεχνικές χρονοδρομολόγησης [ANP03, WS06] που στοχεύουν στην αντιμετώπισή του.

Οι εργασίες [PSC03, PSC05, PDZ00, WWPR04] εστιάζουν στο πρόβλημα της περιττής αντιγραφής δεδομένων σε εξυπηρετητές αποθήκευσης και προτείνουν αλλαγή στην οργάνωση των απομονωτών δεδομένων ώστε να μειωθεί το κόστος της μετακίνησης

δεδομένων ανάμεσα σε αποθηκευτικά μέσα και το δίκτυο.

Μεγάλο μέρος της αποθηκευτικής υποδομής για μεσαίας και μεγάλης κλίμακας υπολογιστικές συστοιχίες στηρίζεται σε παράλληλα συστήματα αρχείων μοιραζόμενου δίσκου, όπως τα GPFS της IBM [SH02], OCFS2 της Oracle [Fas06], Global File System (GFS) της Red Hat [SRO96, PBB+99] και CXFS της SGI [SE04]. Τα περισσότερα βασίζονται σε έναν κατανεμημένο διαχειριστή κλειδωμάτων που ακολουθεί της αρχές του διαχειριστή κλειδωμάτων του ΛΣ VMS [KLS86]. Αυτά τα συστήματα αρχείων εγκαθίστανται είτε πάνω από εξειδικευμένο δίκτυο πρόσβασης (π.χ. Fibre Channel), είτε πάνω από έναν εικονικό μοιραζόμενο αποθηκευτικό χώρο όπως παρέχεται από ένα σύστημα nbd.

Πολλές ερευνητικές εργασίες [TML97, LT96, FLB08, FB05] έχουν επικεντρωθεί στην κατασκευή κατανεμημένης αποθηκευτικής υποδομής, συνδυάζοντας αποθηκευτικά μέσα εγκατεστημένα σε διαφορετικούς κόμβους, καθώς και στην κατασκευή συστημάτων αρχείων που λειτουργούν πάνω από μία τέτοια υποδομή.

Σημαντική ερευνητική κατεύθυνση είναι επίσης οι συσκευές αποθήκευσης βασισμένες σε αντικείμενα (Object-based Storage Devices), όπου το *αντικείμενο* είναι η στοιχειώδης μονάδα αποθήκευσης κι όχι το μπλοκ [osd04, GNA+98]. Το Lustre [Sun08] είναι ένα ευρέως χρησιμοποιούμενο σύστημα αρχείων βασισμένο σε αντικείμενα. Η τεχνική που υλοποιείται στο gmblock μπορεί να εφαρμοστεί στη σχεδίαση ενός αποδοτικότερου εξυπηρετητή αποθήκευσης (Object-Storage Server - OSS) για το Lustre, επιτρέποντας στη διαχείριση των αντικειμένων να γίνεται στη CPU του κόμβου με μικρότερη επιβάρυνση λόγω της κίνησης των δεδομένων ανάμεσα στο αποθηκευτικό μέσο και το δίκτυο. Ομοίως, οι προτεινόμενες τεχνικές μπορούν να εφαρμοστούν για να βελτιώσουν τη δυνατότητα κλιμάκωσης ενσωματωμένων συστημάτων που συνδυάζονται με δίσκους για να υλοποιήσουν αποθήκευση βασισμένη σε αντικείμενα, όπως το σύστημα StorageBlade της Panasas [NSM04, WUA+08].

Το Network File System (NFS) χρησιμοποιείται παραδοσιακά για απομακρυσμένη πρόσβαση αρχείων σε συστήματα UNIX. Η τελευταία αναθεώρησή του, το NFSv4.1, περιλαμβάνει υποστήριξη για παράλληλο NFS (pNFS) [HH05], διαχωρίζοντας το χειρισμό των μετα-δεδομένων των αρχείων από την ανάκτηση των δεδομένων τους από εξυπηρετητές αποθήκευσης. Το gmblock θα μπορούσε να ενταχθεί σε υπάρχουσα εγκατάσταση pNFS ως τρόπος επικοινωνίας με εξυπηρετητές αποθήκευσης σε επίπεδο μπλοκ,

επιτρέποντας σε καθιερωμένους πελάτες pNFS να αλληλεπιδρούν με αποθηκευτική υποδομή παρεχόμενη από το gmblock χωρίς μεταβολές στον κώδικά τους.

Μεγάλο μέρος δικτυακών συσκευών μπλοκ βασίζονται στο TCP/IP, όπως οι GNBD, σε συνδυασμό με το GFS [PBB+99], η Distributed RAID Block Device (DRBD) [Ell07] και το στρώμα Network Shared Disk του GPFS [SH02]. Υλοποιήσεις βασισμένες σε RDMA [KKJ02, LPB04, LYP06, MXPB06, MPB07] απαλείφουν το κόστος της πολύπλοκης στοίβας πρωτοκόλλων και μειώνουν τον αριθμό των απαιτούμενων αντιγράφων, αλλά βασίζονται σε ενδιάμεση αποθήκευση των δεδομένων στην κεντρική μνήμη.

Η διαθεσιμότητα περιοχών μνήμης πάνω στον προσαρμογέα δικτύου έχει οδηγήσει σε ερευνητικές προσπάθειες [KPR02, yKRP05, CKE+05, WYMG09, KPR02, CKE+05, WYMG09] που στοχεύουν στη χρήση τους για προσωρινή αποθήκευση δεδομένων σε εξυπηρετητές αποθήκευσης.

Η παρούσα διατριβή δεν είναι η μόνη που εξερευνά τη δυνατότητα απευθείας μεταφορών δεδομένων ανάμεσα σε αποθηκευτικές συσκευές και προσαρμογείς δικτύου. Το έργο DREAD [Dyd01] περιγράφει ένα μηχανισμό απομακρυσμένου ελέγχου συσκευών SCSI πάνω από απομακρυσμένη πρόσβαση μνήμης μέσω δικτύου SCI. Ο σχεδιασμός του απαιτεί αλλαγές στον οδηγό συσκευής της συσκευής SCSI και επιτρέπει σε ένα μόνο απομακρυσμένο κόμβο να υποβάλλει αιτήσεις Ε/Ε σε δεδομένη συσκευή αποθήκευσης, ενώ εισάγει σημαντική επιβάρυνση στο χειρισμό απομακρυσμένων διακοπών υλικού.

Το Proboscis [Han01, HL02] υλοποιεί ένα σύστημα μοιραζόμενης πρόσβασης επιπέδου μπλοκ πάνω από SCI, όπου οι συμμετέχοντες κόμβοι έχουν διπλό ρόλο, ως εξυπηρετητές υπολογισμού και αποθήκευσης. Επιπλέον, αναφέρει τη δυνατότητα απευθείας μεταφορών ανάμεσα σε απομακρυσμένα αποθηκευτικά μέσα και το δίκτυο πάνω από απεικονίσεις SCI. Μια τέτοια προσέγγιση ωστόσο εισάγει μεγάλες δυσκολίες στο χειρισμό λαθών, καθώς δεν υπάρχει τρόπος εντοπισμού αποτυχημένων προσβάσεων σε απομακρυσμένες θέσεις μνήμης από τη συσκευή αποθήκευσης, ενώ δεν παρέχει συνάφεια με την κρυφή μνήμη σελίδων του πυρήνα.

Η εργασία Off-Processor I/O with Myrinet (OPIOM) [Geo02] ήταν η πρώτη υλοποίηση σε Myrinet απευθείας μεταφορών δεδομένων από τοπικά αποθηκευτικά μέσα στο δίκτυο. Στην πλευρά του εξυπηρετητή, το OPIOM υποστηρίζει πρόσβαση μόνο για ανάγνωση, παρακάμπτοντας τον επεξεργαστή του κόμβου και την κεντρική μνήμη. Ωστό-

σο, απαιτεί αλλαγές στη στοίβα οδηγών SCSI του πυρήνα του Linux, υποστηρίζει μόνο συσκευές SCSI, δεν παρέχει συνάφεια με την κρυφή μνήμη σελίδων του πυρήνα και δεν επιτρέπει την εκτέλεση λειτουργιών απομακρυσμένης εγγραφής.

Το READ^2 [CRU03] ακολουθεί μια διαφορετική προσέγγιση για τη μείωση του κόστους πρόσβασης σε απομακρυσμένα αποθηκευτικά μέσα, προτείνοντας τον έλεγχο των δίσκων με απευθείας εκτέλεση του οδηγού συσκευής πάνω στο Lanai, αντί ως τμήμα του πυρήνα του ΛΣ. Κάθε αίτηση απομακρυσμένης Ε/Ε υφίσταται επεξεργασία στο Lanai, χωρίς εμπλοκή του επεξεργαστή του κόμβου. Μια τέτοια προσέγγιση είναι δύσκολα εφαρμόσιμη, καθώς δεν επιτρέπει ταυτόχρονη πρόσβαση στο μέσο από τον τοπικό κόμβο. Επιπλέον, δεν εκμεταλλεύεται τους ενσωματωμένους οδηγούς συσκευής στον πυρήνα του ΛΣ, απαιτώντας εκτεταμένες μεταβολές στον κώδικα κάθε οδηγού χωριστά, ώστε να είναι εκτελέσιμος στο περιορισμένο περιβάλλον λογισμικού του Lanai, χωρίς να μπορεί να χρησιμοποιήσει τα επίπεδα αφαίρεσης που προσφέρονται από τον πυρήνα του ΛΣ.

Σχετικά με τις βελτιστοποιήσεις στην πλευρά του πελάτη, οι εργασίες [WWP03, YP05] εξερευνούν τη χρήση λειτουργιών RDMA για την υποστήριξη Ε/Ε με χρήση των λιστών διασποράς-συλλογής του συστήματος αρχείων PVFS [CCkL$^+$02]. Επικεντρώνονται στην επιβάρυνση που εισάγουν οι απαιτούμενες λειτουργίες δήλωσης μνήμης στη λειτουργία του συστήματος.

Η εργασία για το ORFA [GP04, GPG04] διερευνά το κόστος πρόσβασης σε κατανεμημένο σύστημα αρχείων μέσω Myrinet, από διεργασία χώρου χρήστη. Διαπιστώνει προβλήματα ανάλογα με αυτά που παρουσιάζονται στο § 0.5.1 και προτείνει επέκταση του GM ώστε να χρησιμοποιεί απευθείας φυσικές διευθύνσεις. Ωστόσο, η υποστήριξη αφορά μόνο απομονωτές συνεχόμενους στη φυσική μνήμη, γεγονός που περιορίζει την εφαρμογή της μεθόδου για μεταφορά δεδομένων μπλοκ σε χωριστές περιοχές της κρυφής μνήμης σελίδων, ή σε απομονωτές χώρου χρήστη διάσπαρτους στη φυσική μνήμη. Η σχεδίασή μας επιτρέπει Ε/Ε από/προς διάσπαρτες περιοχές της μνήμης, με χρήση λιστών διασποράς-συλλογής δεδομένων.

0.7 Συμπεράσματα και μελλοντικές κατευθύνσεις

Τα συστήματα SMP χρησιμοποιούνται συχνά ως δομικές μονάδες για την κατασκευή υπολογιστικών συστοιχιών. Ο διαμοιρασμός πόρων, εγγενής σε τέτοια συστήματα, επιδρά αρνητικά στην απόδοσή τους. Επιπλέον, η λειτουργία του συστήματος Ε/Ε είναι καθοριστική για την συνολική απόδοση του συστήματος κατά την εκτέλεση εφαρμογών απαιτητικών σε δεδομένα.

Ξεκινώντας από αυτές τις παρατηρήσεις, εστιάσαμε σε μηχανισμούς αποδοτικής μετακίνησης δεδομένων από συσκευές αποθήκευσης σε υπολογιστικούς πυρήνες μέσω δικτύου διασύνδεσης, στοχεύοντας σε χαμηλή χρησιμοποίηση CPU και μειωμένο ανταγωνισμό στον διάδρομο μνήμης και τον περιφερειακό διάδρομο. Για το σκοπό αυτό εκμεταλλευτήκαμε χαρακτηριστικά όπως μηχανές DMA και επεξεργαστικές μονάδες που προσφέρουν τα σύγχρονα δίκτυα διασύνδεσης.

Για τη μείωση της επιβάρυνσης που εισάγουν οι απομακρυσμένες λειτουργίες Ε/Ε σε εξυπηρετητές αποθήκευσης, προτείναμε τη χρήση μονοπατιών δεδομένων απευθείας από τα αποθηκευτικά μέσα προς το δίκτυο. Παρουσιάσαμε τη σχεδίαση και υλοποίηση του gmblock, ενός συστήματος nbd που βασίζεται σε τέτοια μονοπάτια, με τρόπο ανεξάρτητο της συσκευής αποθήκευσης, συνδυάζοντας τη μνήμη του προσαρμογέα δικτύου με το μηχανισμό άμεσης Ε/Ε του ΛΣ. Η πειραματική αποτίμησή της επίδοσής του έδειξε σημαντική βελτίωση του ρυθμού απομακρυσμένης Ε/Ε, με μειωμένη παρεμβολή στον υπολογισμό στους επεξεργαστές του κόμβου.

Επιπλέον, προτείναμε μικρές προσθήκες στο υλικό του κόμβου, συγκεκριμένα την προσθήκη μικρών ποσών μνήμης κοντά στο δίκτυο, έτσι ώστε η προσέγγιση του gmblock να είναι εφαρμόσιμη για δίκτυα εκτός Myrinet. Μια τέτοια σχεδίαση μπορεί να επιφέρει σημαντικά οφέλη στην κατανάλωση ενέργειας σε ενσωματωμένα συστήματα, διαχωρίζοντας το μονοπάτι ελέγχου από το μονοπάτι κίνησης των δεδομένων κι επιτρέποντας στον κύριο όγκο της μεταφοράς να συμβαίνει εκτός της κεντρικής μνήμης.

Προτείναμε συγχρονισμένες λειτουργίες αποστολής στο δίκτυο ώστε να είναι δυνατή η επεξεργασία μεγάλων αιτήσεων Ε/Ε στο απευθείας μονοπάτι, χωρίς εμπλοκή του επεξεργαστή του κόμβου για το συντονισμό της διαδικασίας. Η χρήση τους οδηγεί σε επικάλυψη των φάσεων ανάγνωσης από το μέσο και αποστολής στο δίκτυο, με πολλαπλές ροές δεδομένων από διατάξεις RAID.

Στην πλευρά του πελάτη, εκμεταλλευτήκαμε τη δυνατότητα προγραμματισμού του προσαρμογέα δικτύου για να υποστηρίξουμε απευθείας μεταφορές από και προς διάσπαρτες περιοχές της φυσικής μνήμης χωρίς ενδιάμεσα αντίγραφα. Τέλος, μετρήσεις επίδοσης με ρεαλιστικά μετροπρογράμματα πάνω από παράλληλο σύστημα αρχείων έδειξαν ότι το gmblock μπορεί να μειώσει την επίδραση του ανταγωνισμού για μοιραζόμενους πόρους, αρκεί η χρονοδρομολόγηση Ε/Ε στην πλευρά του εξυπηρετητή να παρέχει ικανοποιητικό ρυθμό μεταφοράς από τα αποθηκευτικά μέσα όταν χρησιμοποιούνται από πολλούς πελάτες ταυτόχρονα.

Στη συνέχεια, συζητάμε κατευθύνσεις για μελλοντικές επεκτάσεις στην παρούσα δουλειά.

Πρόσφατη έρευνα στο πεδίο των εξυπηρετητών αποθήκευσης [PFB09] εστιάζει σε αποδοτικές τεχνικές χρονοδρομολόγησης Ε/Ε για μεγάλο αριθμό ταυτόχρονων ρευμάτων δεδομένων. Ανάλογες τεχνικές θα μπορούσαν να χρησιμοποιηθούν σε συνδυασμό με το gmblock για τη βελτίωση της απόδοσης του συστήματος αποθήκευσης, αν και θα έπρεπε να προσαρμοστούν ώστε να χρησιμοποιούν το περιορισμένο ποσό μνήμης που παρέχει ο προσαρμογέας δικτύου και να αποθηκεύουν δεδομένα σε κρυφές μνήμες στην πλευρά των πελατών.

Ως μακροπρόθεσμο στόχο για μελλοντική δουλειά βλέπουμε την επέκταση των μηχανισμών διαχείρισης μνήμης του ΛΣ έτσι ώστε να αναγνωρίζουν την ύπαρξη χωριστών περιοχών μνήμης στο σύστημα, με διαφορετικό ρόλο: ορισμένες κοντύτερα σε υπολογιστικούς πυρήνες, ορισμένες κοντύτερα στο δίκτυο. Η τρέχουσα σχεδίαση κατασκευάζει πλαίσια σελίδων για τη μνήμη του προσαρμογέα δικτύου, ωστόσο αυτά σημειώνονται ως δεσμευμένα· καθιστούν δυνατή την πραγματοποίηση άμεσης Ε/Ε, αλλά δεν υφίστανται διαχείριση από το υποσύστημα μνήμης του πυρήνα. Ο στόχος μας είναι η επέκταση του πυρήνα ώστε να υποστηρίζει την χρήση τους ως κρυφή μνήμη σελίδων, η οποία θα κατανέμεται ανάμεσα στην κύρια μνήμη του συστήματος και σε μνήμες σε προσαρμογείς του δικτύου. Ο εμπλουτισμός της σημασιολογίας λειτουργιών δέσμευσης μνήμης θα επιτρέψει σε εφαρμογές να υποδεικνύουν τι είδους χρήση αναμένεται να έχει ένας απομονωτής δεδομένων, ζητώντας να απεικονιστεί ένα αρχείο σε μνήμη κοντά στο δίκτυο.

Σε επίπεδο αρχιτεκτονικής, αυτό μπορεί να επιτευχθεί με την εξαγωγή περιοχών μνήμης από τις συσκευές στο χώρο διευθύνσεων του PCI.

Η επέκταση της κρυφής μνήμης σελίδων ώστε να διαθέτει τη λειτουργικότητα που περιγράφεται θα επιφέρει σημαντική βελτίωση στη λειτουργία του συστήματος, επιτρέποντας την προανάκτηση δεδομένων στην πλευρά του εξυπηρετητή και υποστηρίζοντας ιεραρχική οργάνωση των διαθέσιμων περιοχών μνήμης: όταν η μνήμη στον προσαρμογέα δεν επαρκεί, το σύστημα μεταπίπτει σταδιακά στη χρήση απομονωτών στην κύρια μνήμη. Υπάρχει ένας συμβιβασμός ανάμεσα στην υποστήριξη μεγαλύτερου συνόλου εργασίας στην κρυφή μνήμη και τη σταδιακή αύξηση της πίεσης στο διάδρομο μνήμης του κόμβου.

1 Introduction

1.1 Motivation

Clusters built out of commodity components have become prevalent in the supercomputing sector as a cost-effective solution for building scalable parallel platforms. The use of clustered systems has expanded to provide a high-performance computing infrastructure for various disciplines; clusters have been powering such diverse workloads as cosmological simulations in astrophysics, weather forecast models in meteorology, vehicle collision tests in the car industry, graph algorithms for web search engines, and data mining applications for business intelligence.

Symmetric Multiprocessors (SMPs) of multicore chips (CMPs), are commonly used as building blocks for scalable clustered systems, when interconnected over a high-bandwidth, low-latency communications infrastructure, such as Myrinet [BCF+95], Quadrics [PcFH+01] or Infiniband [Inf00].

Fig. 1.1 displays the typical layout of an SMP cluster node. A small number of processors are interconnected over a shared Front-Side Bus to a memory controller, commonly called the Northbridge in Intel-based designs, which provides access to a number of memory modules (Fig. 1.1). The memory controller is part of a chipset, also containing bridges to one or more peripheral buses, commonly PCI or PCI-X. On the peripheral bus lie a number of I/O devices; the most important are the Network Interface Card (NIC), which enables communication with the rest of the cluster, and storage controllers, e.g., RAID cards for access to mass storage devices.

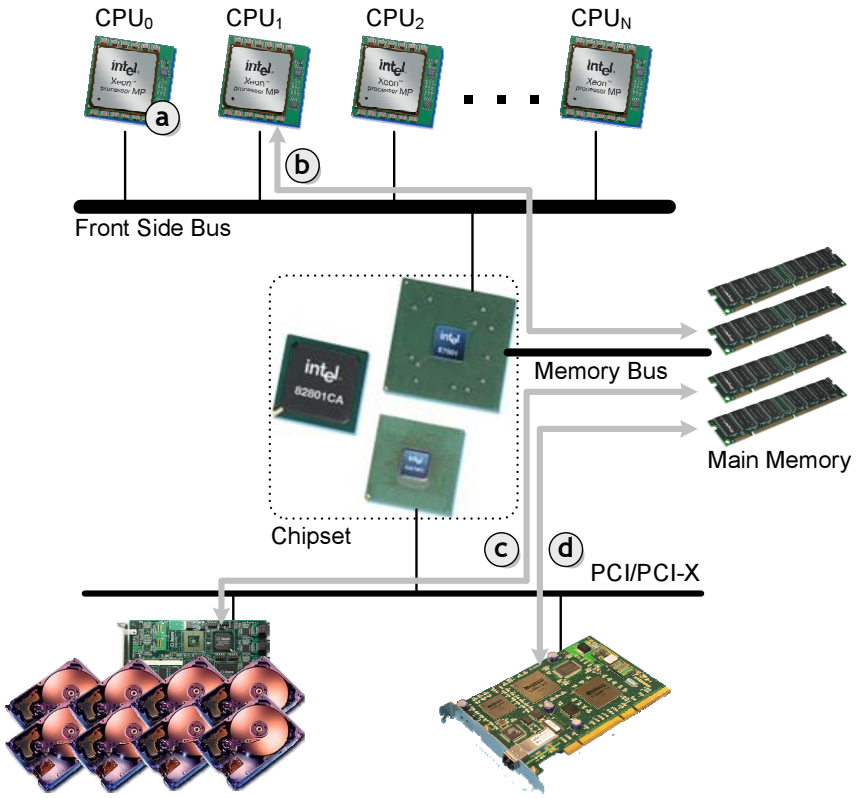


Figure 1.1: *Typical layout of an FSB-based SMP system*

The most important architectural characteristic of an SMP system is *resource sharing* at multiple levels. Inside a single chip, multiple cores often share access to some levels of the cache hierarchy. Leaving the chip, all processors share bandwidth on the Front-Side Bus and the bus to main memory. At the same time, I/O devices may access main memory independently of the CPUs, using Direct Memory Access (DMA) techniques. Thereby, a device takes control of the peripheral bus (*bus mastering*) and issues memory access requests, which are forwarded through the bus bridge to the Northbridge, to be serviced by main memory. Thus, I/O devices share bandwidth on their peripheral bus and also contend with the CPUs for access to main memory. Finally, processes running on the CPUs share access to mass storage devices and bandwidth to the cluster interconnect.

Resource sharing simplifies the design of the system and facilitates its programming, however it may have significant impact on overall system performance. The extent of resource sharing and its performance impact depends on application behavior. Applications with excellent cache locality will spend a significant fraction of their execution

time inside the processor, working with data in the cache (Fig. 1.1, point (**a**)). On the other hand, data-intensive applications pose significant load on the path to main memory (Fig. 1.1, path (**b**)). Such applications are those with increased data to computation ratio, e.g., sparse scientific computations [GKA+09], graph algorithms and database transaction processing. When the dataset to be processed no longer fits in main memory, as is the case for most enterprise workloads, execution efficiency becomes sensitive to the performance of the path to storage. In clustered systems, access to a shared storage system is commonly provided over a storage area network or over the cluster interconnection, as described in greater detail below and in Section 2.1. Thus, such applications are sensitive to the performance of paths (**c**), (**d**) in Fig. 1.1.

The need for high-performance I/O is also becoming prevalent outside the HPC domain. Traditional data-intensive applications such as Online Transaction Processing and Web search engines are complemented by applications from areas such as social networking, video streaming, and file sharing. A recent report by the International Data Corporation [GCM+08] estimates the amount of data generated and stored worldwide to reach 1800EB (18×10^{20} bytes) by 2011. A significant fraction of the data produced will be accessible remotely. This trend is exemplified by the emergence of ubiquitous, low-power embedded storage devices for network-based access to storage, based on a System-on-Chip (SoC) architecture, e.g., [Mar].

At the same time, advances in microprocessor design have been fueling the trend for multiple cores per processor die. The number of cores has been increasing and will most probably continue to grow in the years to come, thus providing substantial processing power on a single chip. Having this amount of processing power available poses tremendous load on the I/O subsystem, which becomes a decisive factor in performance. For applications to scale with the number of cores, the I/O subsystem must meet the challenge of feeding data fast enough to keep all cores busy. [Gur09]

The increase in available computational capacity, combined with the emergence of applications with heavy data-processing demands on SMP clusters, shifts the focus to network I/O. We need efficient mechanisms for *transporting* large datasets efficiently between compute cores and secondary storage, over an interconnection network, with minimal overhead.

Systems for block-level storage sharing over the interconnection network are commonly

used to provide for scalable, yet cost-effective deployment of various services in high-performance clustered environments. Such services include shared-disk parallel filesystems for HPC applications, shared-disk parallel databases, and shared storage pools for live Virtual Machine (VM) migration in virtualized data centers.

A block-level storage sharing system enables a number of clients to access storage devices on a remote server as if they were local. Block read and write requests are encapsulated in network messages to the storage server, where they are passed to the local block device (Fig. 1.2). When the block operation completes, the server returns the resulting data over the network. Throughout this dissertation, we refer to such systems also as "network block device" systems, or *nbd systems*; storage exported by a storage server appears to the client as a block device accessible over the network.

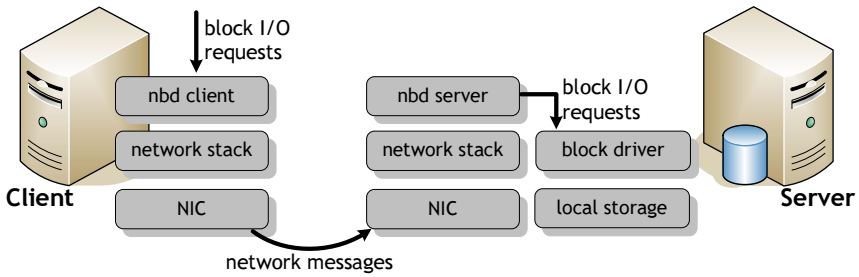


Figure 1.2: *Operating principle of an nbd system*

An ideal nbd system provides a very thin, low-overhead layer that allows remote use of storage media with performance close to that of local access. Furthermore, it should scale with the number of storage subsystems and network interfaces, without imposing significant load on the storage servers and clients. In SMP clusters, the demand for low-overhead access translates to *reduced CPU load* and *minimal interference* due to the use of shared resources, i.e., bus bandwidth on the shared memory and peripheral buses; bus capacity emerges as a valuable resource, limited under several realistic execution scenarios.

Even when an SMP cluster node is viewed in isolation, memory contention can lead to substantial slowdown; two or more memory-intensive applications, each on its own dedicated processor, may interfere with each another during memory access, on path **(b)**, increasing the average cost per memory transaction. Various works in the literature have focused on dynamic monitoring and scheduling of processes to mitigate the impact of resource sharing and deliver predictable performance [Bel97, LVE00, ANP03]. More

information can be found in Chapter 6.

Peripheral devices, such as local storage controllers and Network Interface cards, exacerbate the problem of resource contention, since they also contend for access to main memory (Fig. 1.1, paths **(c)**, **(d)**). The evolution of cluster interconnection technology has brought considerable increases in their link rate, currently into the 10-40Gbps range, which is a significant portion of the available memory bandwidth. This leads to network I/O operations interfering with local computation on the system processors [Sch03], increases memory pressure and has noticeable performance impact [KK05, KK06].

Since the efficiency of the I/O infrastructure is pivotal to overall performance, the design of an nbd system needs to minimize the overhead of remote data access; CPU involvement and contention on the path between processing cores and remote storage aggravates this overhead significantly.

However, current nbd implementations are suboptimal in that regard; Often, they invoke heavy host CPU-based processing, being based on TCP/IP for the transfer of block data. More importantly, they treat memory as a centralized resource and make use of data paths that cross the memory and peripheral buses multiple times, even when employing advanced interconnect features such as user level networking and remote DMA (RDMA) [KKJ02, LPB04, LYP06]. Thus, they impose high host overhead and their performance is limited due to bus saturation. The current situation is analyzed in greater detail in Section 2.1 and Chapter 6.

1.2 Contribution

This work explores the implications of CPU, memory bus and peripheral bus contention in SMP nodes used as commodity storage servers. We study the data movement in a block-level storage sharing system over Myrinet and show how its performance suffers as the storage subsystem, the network and local processors all compete for access to main memory and peripheral bus bandwidth. To alleviate the problem, we explore techniques for building efficient data paths between the storage medium and the network on the server side, and the network and processing cores on the client side. We focus on system-level software optimizations to limit the impact of contention and improve system throughput.

To minimize CPU overhead and reduce the load due to redundant data movement on storage servers, we look into how an nbd system can exploit a number of relevant features provided by current cluster interconnects. Such features include various degrees of programmability of their Network Interfaces (NIs), and the possibility of offloading parts of protocol processing to dedicated cores and memories close to the network.

We present the design and implementation of gmblock, a block-level storage sharing architecture over DMA- and processor-enabled cluster interconnects that is built around a short-circuit data path between the storage subsystem and the network. Our prototype implementation uses Myrinet, allowing direct data movement from storage to the network without any host CPU intervention and eliminating any copies in main memory. This alleviates the effect of resource contention, increases scalability and achieves an up to two-fold increase in performance compared to standard approaches. Moreover, this means that the storage server no longer needs to be used exclusively for servicing I/O requests; it can have a dual role, as a compute and storage node, since remote I/O follows a disjoint path and does not interfere with computation on the local CPUs. The design of gmblock enhances existing OS and user level networking abstractions in order to construct the proposed data path, rather than on relying on architecture-specific code changes. Thus, it is independent of the actual type of block device used, can support both read and write access safely, and maintains the process isolation and memory protection semantics of the OS.

Experimental evaluation of the base gmblock implementation shows that although it works around memory and peripheral bus bandwidth limitations effectively, its performance lags behind the limits imposed by raw disk and network bandwidth. By breaking down request processing in phases and studying each individually, we find this is due to an interplay between the data movement characteristics of real-world, RAID-based storage systems and memory limitations of the Myrinet NIC. To better adapt gmblock to the inherent parallelism in request processing by RAID storage and to enable processing of large I/O requests with reduced host CPU overhead, we propose a new class of send operations over Myrinet, which support *synchronization*: their semantics allow the network transfer of block data to progress in a controlled way, concurrently with disk I/O, overlapping disk with network I/O for a single block request. A working prototype, based on custom modifications to Myrinet's GM message-passing middleware, shows significant improvement for streaming I/O, compared to the base version of gmblock.

We study data movement on the client side and propose techniques which exploit NIC programmability to support zero-copy block transfers to application buffers dispersed in physical memory with minimal host CPU involvement. Combined with the proposed server-side data path, our system can support end-to-end zero-copy block transfers, from remote disk to local memory. We deploy a shared-disk parallel filesystem, Oracle's OCFS2 on top of this infrastructure to evaluate the performance with application workloads. We find that using the direct I/O path generally leads to performance improvement, provided incoming I/O requests can be scheduled efficiently, so that the disk subsystem does not become the bottleneck.

The contribution of this thesis can be summarized as follows:

- We propose direct I/O paths between storage devices and the network, to alleviate the effect of memory and peripheral bus contention on commodity storage servers. We show how such paths can be built in a block device-independent manner, exploiting NIC-based memory areas and generic OS mechanisms for direct I/O.

- We present gmblock, a prototype implementation over Myrinet. Experimental evaluation shows it eliminates memory and peripheral bus contention, delivering significant improvements to remote read/write I/O bandwidth. We demonstrate how bypassing main memory enables local computation to progress with negligible interference from remote I/O.

- We discover limitations in hardware components of the system which reduce the efficiency of peer-to-peer data transfers. We show how our design can work around these limitations by employing an alternate data path using intermediate buffers on the peripheral bus while still bypassing main memory.

- We propose architectural modifications to make the approach of gmblock applicable to interconnection technologies other than Myrinet. We argue for changes to the semantics of memory allocation in the OS, to support management of distinct memory areas: some closer to processors, some closer to the Network Interface.

- We propose synchronized send operations as an enhancement to the semantics of Myrinet's message-passing layer. They exploit NIC programmability to improve

handling of larger requests, with intra-request overlapping of network and block I/O, without host CPU involvement. Their operation is adapted to the multiple-stream nature of request servicing by RAID-based storage.

- We present client-side optimizations to support zero-copy block transfers between the network and application I/O buffers. Integration with gmblock's optimized data path on the server side allows for end-to-end zero-copy transfers between client-side userspace buffers and remote storage. To the best of our knowledge, this is the first such implementation.

- We explore the tradeoffs of using the proposed data path with regard to server-side caching and prefetching, by deploying a production-quality shared-disk parallel filesystem over our prototype implementation and evaluating its performance with various workloads.

1.3 Outline

This dissertation is organized as follows: In Chapter 2 we present the basic operating principles of nbd systems and how their deployment can enable scalable clustered storage in various contexts. Then, we discuss briefly the core concepts of user level networking and its implementation in Myrinet/GM and conclude with a brief description of the Linux block layer, to highlight the parts of its functionality pertinent to our work. Chapter 3 describes the proposed server-side data path and its implementation in gmblock. Chapter 4 concerns our work on extending GM with synchronized send operations. In Chapter 5 we explore client-side optimizations and evaluate the performance of a parallel filesystem deployment over our storage infrastructure with realistic workloads. In Chapter 6 we compare our approach to related work in the literature, while Chapter 7 summarizes our conclusions and provides directions for future work.

2 Background

In this chapter, we provide background information on the hardware and software environment targeted by this work. Initially, we discuss the basic operating principles behind nbd systems and present common usage scenarios for their deployment in clusters. The next section contains a short introduction to the essentials of user level networking and its implementation in Myrinet/GM, to gain insight on the infrastructure used as the communications substrate of the nbd system which is the focus of this work. The chapter concludes with a short description of the Linux block layer, to understand the basic concepts behind block device management in a modern Operating System, and thus the context in which the client-side portion of an nbd system operates.

2.1 nbd systems and applications

The need for shared block-level access to common storage arises often in clustered environments. Some of the most common scenarios include *(a)* the deployment of shared-disk parallel filesystems, *(b)* support of parallel databases based on a shared-disk architecture, such as Oracle RAC, and *(c)* virtualized environments, where disk images of virtual machines are kept in common storage, so that live migration of them among VM containers is possible.

In the first case, parallel filesystems are deployed to meet the I/O needs of modern HPC applications. They allow processes running on cluster nodes to access a common filesystem namespace and perform I/O from and to shared data concurrently. There are various implementations of cluster filesystems which focus on high performance, i.e., high

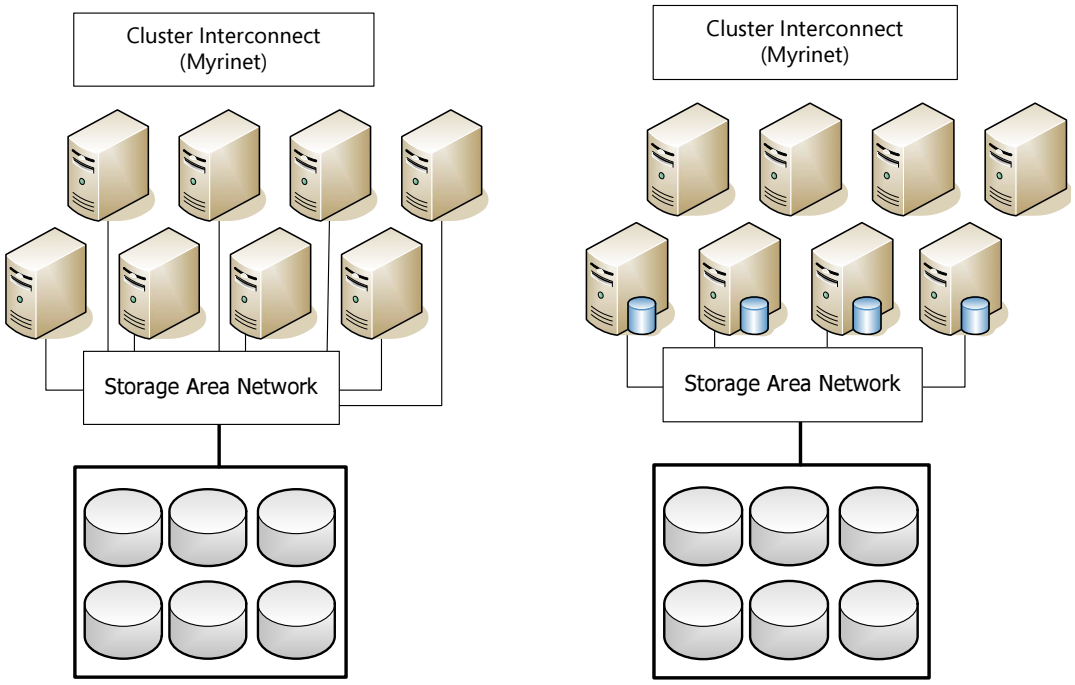
aggregate I/O bandwidth, low I/O latency and high number of sustainable I/O operations per second, as multiple clients perform concurrent access to shared data. At the core of their design is a *shared-disk* approach, in which all participating cluster nodes are assumed to have equal access to a shared storage pool. The shared-disk approach is followed by filesystems such as IBM's General Parallel File System GPFS [SH02], Oracle's OCFS2, Red Hat's Global File System (GFS) [SRO96, PBB+99], SGI's Clustered XFS [SE04], and the VERITAS Cluster File System [Sym], which aim to provide a high-performance parallel filesystem for enterprise environments.

In the second case, instances of a shared-disk parallel database execute on a number of cluster nodes and need concurrent access to a shared disk pool, where table data, redo logs and other control files are kept.

Finally, in the case of virtualized environments, a virtual machine runs on a cluster node acting as a VM container and performs raw block-level access to a storage volume, which it treats as a directly-connected hard drive. For reasons of load balancing and maintainability, it is desirable to be able to live migrate the VM among VM containers. Thus, the storage volumes must be viewable by all VM containers, to ensure uninterrupted access by a VM to the virtual disk images associated with it when migration occurs.

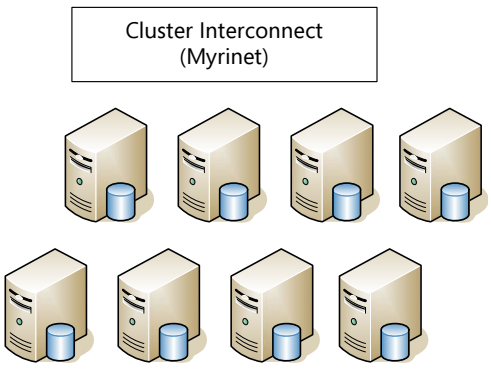
Traditionally, the requirement that all nodes have access to a shared storage pool has been fulfilled by utilizing a high-end Storage Area Network (SAN), commonly based on Fibre Channel (as in Fig. 2.1(a)). An SAN is a networking infrastructure providing high-speed connections between multiple nodes and a number of hard disk enclosures. The disks are treated by the nodes as Direct-attached Storage, i.e., the protocols used are similar to those employed for accessing locally attached disks, such as SCSI over FC.

However, this storage architecture entails maintaining two separate networks, one for access to shared storage and a distinct one for cluster communication, e.g., for message-passing between MPI peer processes. This increases the cost per node, since the SAN needs to scale to a large number of nodes and each new member of the cluster needs to be equipped with an appropriate interface to access it (e.g., an FC Host Bus Adapter). Moreover, while the number of nodes increases, the aggregate bandwidth to the storage pool remains constant, since it is determined by the number of physical links to the storage enclosures. Finally, to eliminate single points of failure (SPOFs) on the path to



(a) All nodes on SAN

(b) Storage nodes on SAN



(c) Only local storage

Figure 2.1: *Interconnection of cluster nodes and storage devices*

the shared storage pool, redundant links and storage controllers need to be used, further increasing total installation costs.

A hybrid approach can address these problems; shared-disk filesystems are commonly deployed in such way that only a small fraction of the cluster nodes is physically connected to the SAN ("storage" nodes), *exporting* the shared disks for block-level access by the remaining nodes, over the cluster interconnection network. In this approach (Fig. 2.1(b)), all nodes can access the shared disk pool, by issuing block read and write requests over the interconnect. The storage nodes receive the requests, pass them to the storage subsystem and eventually return the results of the operations back to the client node. Taken to extreme, this design approach allows shared-disk filesystems to be deployed over shared-nothing architectures, by having each node contribute part or all of its locally available storage (e.g., a number of directly attached Serial ATA or SCSI disks) to a *virtual*, shared, block-level storage pool (Fig. 2.1(c)). This model has a number of distinct advantages: first, aggregate bandwidth to storage increases as more nodes are added to the system; since more I/O links to disks are added with each node, the performance of the I/O subsystem scales along with the computational capacity of the cluster. Second, the total installation cost is drastically reduced, since a dedicated SAN remains small, or is eliminated altogether, allowing resources to be diverted to acquiring more cluster nodes. These nodes have a dual role, both as compute and as storage nodes.

The cornerstone of this design is the *network disk sharing* layer, implemented in a client-server approach. The main principle behind its operation is portrayed in Fig. 2.2. It runs as a server on the storage nodes, receiving requests and passing them transparently to a directly-attached storage medium. It also runs as a client on cluster nodes, exposing a block device interface to the Operating System and the locally executing instance of the parallel filesystem.This way, it can service block I/O requests by exchanging data with a suitable server instance over the interconnection network. There are various implementations of such systems, facilitating block-level sharing of storage devices over the interconnect. GPFS includes the NSD (Network Shared Disks) layer, which takes care of forwarding block access requests to storage nodes over TCP/IP. Traditionally, the Linux kernel has included the NBD (Network Block Device) driver [1] and Red Hat's

[1] nbd in all small letters will be used to denote generic client-server implementations for network sharing of block devices. NBD in all capital letters denotes the TCP/IP implementation in the Linux kernel.

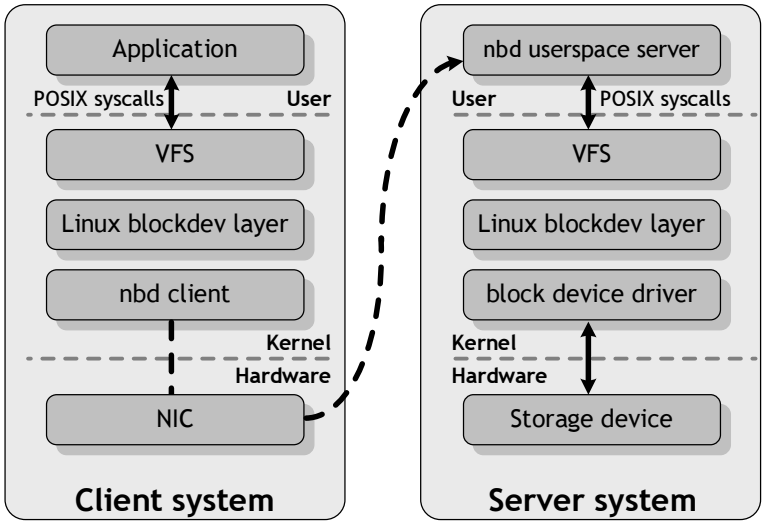


Figure 2.2: *Generic nbd system*

GFS can also be deployed over an improved version called GNBD.

However, all of these implementations are based on TCP/IP. Thus, they treat all modern cluster interconnects uniformly, without any regard to their advanced communication features, such as support for zero-copy message exchange using DMA. Employing a complex protocol stack residing in the kernel results in very good code portability but imposes significant protocol overhead; using TCP/IP-related system calls results in frequent data copying between userspace and kernelspace, increased CPU utilization and high latency. Moreover, this means that less CPU time is made available to the actual computational workload executing on top of the cluster, as its I/O load increases.

On the other hand, cluster interconnects such as Myrinet and Infiniband are able to remove the OS from the critical path (*OS bypass*) by offloading communication protocol processing to embedded microprocessors onboard the NIC and employing DMA engines for direct message exchange from and to userspace buffers. This leads to significant improvements in bandwidth and latency and reduces host CPU utilization dramatically.

As explained in greater detail in the chapter on related work, there are research efforts focusing on implementing block device sharing over such interconnects and exploiting their Remote DMA (RDMA) capabilities. However, the problem still remains that even though the protocol layer may be greatly simplified and the number of copies reduced, the data follow an unoptimized path. Whenever block data need to be exchanged with

a remote node they first need to be transferred from the storage pool to main memory, then from main memory to the interconnect NIC. These unnecessary data transfers impact the computational capacity of a storage node significantly, by aggravating contention on shared resources as is the shared bus to main memory and the peripheral (e.g., PCI) bus. Even with no processor sharing involved, compute-intensive workloads may suffer significant slowdowns since the memory access cost for each processor becomes significantly higher due to contention with I/O on the shared path to memory.

2.2 User level networking

2.2.1 Basic concepts

Recent advances in high-performance networks have brought significant increases in the available bandwidth at the physical layer. Available solutions have evolved to provide bandwidth in the 2-40Gbps range. At the same time, hardware and wire latencies have decreased and are in the 0.3–1.0μs range, depending on the size of the network. Delivering this kind of performance to the application layer has proven difficult, however, mainly due to the use of complex in-kernel protocol stacks, such as TCP/IP. In this case most of the latency is in *software*; the application invokes system calls in order to manipulate the interconnect, which brings the OS in the critical path of communication. Typically, this involves expensive CPU mode switching when trapping into the kernel and CPU-based data copying between userspace and kernelspace buffers.

To mitigate the overhead of OS involvement, most modern cluster interconnects, such as SCI [Hel99], Quadrics [PcFH+01], Infiniband [Inf00] and Myrinet [BCF+95] employ *user level networking* [BRB98b] (ULN) techniques in order to remove the OS from the critical path. In this model, the application process is allowed to control the Network Interface (NI) directly. Since the OS is no longer invoked for communication, its role is undertaken by a combination of application level libraries and firmware executing on the NIC; the capability for data exchange between the two entities is established by privileged code inside an OS kernel module (Fig. 2.3).

The application is allowed direct control of the NI by mapping part of the NI register or memory space into its own virtual memory; following that, it uses unprivileged

load/store instructions into the relevant VM segments in order to communicate.

Removing the OS and CPU from the critical path of communication means certain functionality is implemented on the NIC itself, which undertakes parts of the user level networking protocol. This requires an *intelligent* NIC, with a certain degree of programmability. However, exactly how programmable a NIC needs be is a matter of debate and depends on the design of the user level networking protocol employed.

There is a number of important issues to be considered in designing network architectures supporting operations in userspace, creating an array of possibilities. Past research in the field has focused on the design and implementation of user level networking protocols, each of which offers different programming semantics and make different performance tradeoffs. Some of the design considerations are:

Programming Semantics ULN Protocols differ on the programming semantics offered to applications and can be divided in three categories: shared-memory protocols, send/recv-based message passing protocols and protocols based on 1-sided remote memory get/put operations. Shared-memory protocols include SISCI [GAB+], VMMC and VMMC-2 [DBC+97]. Active Messages II [CMC98], Fast Messages [PLC95], RWC's PM [TOHI98], LFC [BRB98a], U-Net [vEBBV95], Hamlyn [BJM+96] and BIP [PT97] offer explicit message passing with receive-side matching. Finally, interfaces offering 1-sided Remote DMA (RDMA) semantics include the uDAPL (User Direct Access Programming Library) [LPSS03] for RDMA-enabled interconnects and implementations of the Infiniband Verbs specification, itself an evolution of the Virtual Interface Architecture [vEV98, DRM+98].

Data movement To communicate, message data needs to be moved from application buffers—usually residing in host RAM—to some part of NI local memory or other staging buffer. This can be done using *Programmed I/O* (PIO); the application uses loads or stores to transfer all of the message content to mapped NI memory. The downside is that this method wastes host CPU cycles for data movement, may pollute the host CPU cache and incurs the overhead of a large number of peripheral bus transactions, since data is transferred in units of one or two words at a time. A possible optimization is *write-combining*: the results of store

instructions are stored in special write-combining buffers on the CPU so they can be written to NI memory in a single transaction.

Modern cluster interconnect NICs offload data movement from the CPU by using *Direct Memory Access* (DMA) engines to access message data in host RAM in large bursts over the peripheral bus. This allows the creation of *zero-copy* protocols, where the CPU is removed completely from the critical path and is left free to perform computation, enabling computation-to-communication overlapping. However, DMA initialization incurs significant DMA startup overhead, so there is a tradeoff; for small enough messages, PIO with write-combining may outperform DMA.

Thus, some implementations (FM, LFC) perform PIO exclusively while others are DMA-based (PM, VMMC-2, U-Net). A hybrid approach is also possible (AM-II, Hamlyn, BIP), where PIO is only used for small messages, even merging message data into the request descriptor being written into NI memory.

Another issue rising from the use of DMA is the need for *memory registration*. Message buffers lie in application VM space and may have been swapped out by the OS when an application issues a send request or an incoming message arrives. DMA operations refer directly to the physical address space. Since the kernel is no longer in the critical path there is no guarantee that the requested VM page actually exists in physical memory or that it will not be swapped out in the middle of the transfer, leading to memory corruption. In the case of PIO, the first application reference to a swapped out page would cause a page fault, trap into the kernel and cause it to be fetched back into RAM.

In the case of DMA, the relevant pages need to be *wired* or *pinned*, so that they are marked as unswappable by the OS. Memory *registration* with the OS is an expensive operation that requires a system call, thus it must be avoided in the critical path of communication. A simple solution would be to register all the needed buffers at application initialization, which is not always possible, however. First because there is an upper limit imposed by the OS on the number of pinned pages to ensure system stability. Second, because the addresses of the buffers involved in network operations may not be known beforehand due to the semantics of the upper layers, e.g., when implementing MPI over a ULN-based interconnect.

Solutions include using fixed, pre-allocated DMA staging areas and incurring the overhead of memory copying (as AM-II and Hamlyn do), or managing a cache of pinned-down memory areas with performance dependent on the degree of buffer reuse [TOHI98].

Address Translation The application uses references in a virtual address space to specify message buffers for communication operations. However, the NI is a physical device; at some point, it needs to know the actual physical addresses of the relevant pages in RAM, in order to setup the needed DMA transactions. When using PIO, the processor's MMU does the necessary translation whenever a load or a store takes place. In case of DMA, if the kernel is in the critical path, it can consult the corresponding page table entries. Hence, in a ULN scheme, an address translation mechanism becomes necessary. There are many different alternatives to implementing such scheme which depend on the hardware resources available on the NI, the data movement modes supported by the protocol and whether memory protection is required. Design decisions have significant performance implications as well [SH98].

There are two main issues: who will be responsible for performing virtual to physical translations, the CPU or the NI itself, and who will be responsible for handling misses in the corresponding cache.

If protection can be sacrificed, then the application can perform CPU-based address lookups using an interface exposed by the kernel, then provide the NIC with physical addresses directly (this approach is followed by LFC and BIP).

A simple solution would be to use special DMA areas for transfers, for which a permanent mapping may be setup on the NIC. Then, all requests can refer to offsets in this area. Although this approach is very simple to implement and requires very little hardware on the NIC, it is very limited because every transfer incurs the cost of memory copying in and out of these areas (AM-II, Hamlyn, FM in the receive path).

A better solution is to keep a small translation cache for virtual to physical address mapping on NI memory, and perform address lookup on the NIC using either special hardware or a software implementation running on a programmable microcontroller (e.g., the Lanai on Myrinet NICs). However, this translation cache

can only hold a limited number of translations and cannot cover all of the user addressable space, so the issue of handling misses arises.

Handling of misses can be undertaken by the NI itself, by the host CPU, or it may happen in a combined manner, with various degrees of host CPU involvement. Quadrics QsNet^{II} is at one end of the spectrum; its NIC is based on the Elan thread processor which features its own MMU. However, extensive kernel patching is needed to keep the Elan4 MMU synchronized with the host-based pagetables. The NIC processor is treated as a peer to the host CPU, propagating pagetable updates to it as new VM areas are created and destroyed. This approach has the advantage of supporting zero-copy communication from pageable memory, but this comes at the expense of a fully-fledged microprocessor on the NIC, and heavy kernel patching.

At the other end, the NIC may interrupt the host CPU whenever a valid translation cannot be obtained from its cache, causing a trap to the device driver's handling routine. The driver can then wire down the relevant pages, update the NIC-based cache and allow the operation to continue (U-Net/MM). This incurs the overhead of trapping into the OS inside the critical path, however.

Other approaches allow the application to use a kernel interface to pin down and manage references to virtually addressed message buffers, which are then cached on NI memory (VMMC-2). Similarly, Myrinet/GM requires explicit system calls to the kernel in order to register communication buffers with GM-specific pagetables held in kernel memory. The NIC keeps an LRU cache of these tables in NIC memory and updates its contents in the critical path, DMAing the necessary translation data whenever there is a miss.

Protection To be able to integrate a ULN architecture in a real-world operating environment, it is imperative that it respects the memory protection and process isolation semantics of the host OS. This means that no process will be allowed to read or write to memory regions belonging to other processes or access the contents of their messages. When using a protocol stack inside the kernel, this is ensured via a combination of CPU and OS support.

Some ULN schemes assume only trusted processes will be able to use the interface and do not offer protection (FM, LFC, BIP).

To be able to support protection, the protocol needs to provide a *virtualized* view of the interface to the application layer with multiple processes issuing network requests simultaneously. Since the kernel is no longer involved in the critical path, the NI itself needs to verify the validity of these requests.

The common solution is to implement virtual communication endpoints in NIC memory, which are then mapped independently by processes. Mapping virtual memory regions to I/O regions is a privileged operation and is done at interface initialization time using the network's device driver, ensuring protection. After that, a process can only access request descriptors and other resources of its own virtual interface.

The issue of protection is closely related to that of address translation, since the virtual to physical translation step ensures that any invalid references are caught and handled appropriately.

Network endpoints are commonly created via static allocation of the available NI memory, which means the maximum number of concurrent contexts is limited by its size. Another approach (implemented in AM-II) is to only have a limited number of endpoints cached in NI memory and dynamically swap endpoints between NI memory and host memory, as appropriate, suffering the corresponding performance hit.

Control transfer and completion notification A way is needed for applications to be notified whenever a network operation completes, e.g., a send operation completes successfully or a new incoming message has been received. Since the design of ULN architectures is driven by the need for low latency, most eschew interrupt-based host notification for *polling*. Flags or event queues residing in NI or host memory are used to notify the application of network events.

The application polls the queue at regular intervals. Storing the completion queues in NI memory may lead to high volume of I/O operations. Keeping them in host memory is very efficient if the architecture supports coherent DMA; this way memory contents are cached and the corresponding lines are invalidated when the NI DMAs a new descriptor into the queue. Otherwise, these memory areas must be uncacheable and the application will always go to main memory in order to poll the queue.

Polling has the disadvantage of wasting CPU cycles and is avoided in multiprogrammed environments. Thus, most ULN architectures support interrupts in addition to polling; the application can enter the kernel and block, releasing the CPU, but incurring the latency of interrupt handling in the critical path. When a new event happens, the NIC interrupts the host, the device driver interrupt handler runs, and wakes up any waiting processes.

Finally, hybrid approaches may be used, in which the application spins for a short time, while the NIC has interrupts disabled, then enters the kernel, where it blocks and interrupts are enabled on the NIC.

2.2.2 Implementation in Myrinet/GM

Myrinet is a low-latency, high-bandwidth interconnection infrastructure for clusters. Two generations of Myrinet are currently available: Myrinet-2000 and Myri-10G. The physical layer of Myrinet-2000 uses full-duplex, point-to-point 2+2Gbps fiber links. Nodes are interconnected over crossbar switches, in a Clos topology. Myrinet uses source routing: the network is mapped, so that each participating node knows how to reach every other node using up*/down* routing and every packet contains the full route to its destination, as a series of ports to be traversed at each switch. This way, low-latency cut-through switching is possible. Myri-10G, the newer generation, is based on the same physical layer as 10-Gigabit Ethernet to increase the link rate to 10Gbps, and can use either the source-routed Myrinet protocol or 10-Gigabit Ethernet at the Data Link Layer.

To reduce the overhead of OS involvement, Myrinet employs user level networking techniques [BRB98b] in order to remove the OS from the critical path of communication. In this model, an application process is allowed to control the Network Interface (NI) directly; since the OS is no longer invoked for communication, its role is undertaken by a combination of application level libraries and firmware executing on the NIC, while data exchange between the two is setup by privileged code inside an OS kernel module. The Myrinet software stack is displayed in Fig. 2.3.

The application is granted control of the NI by mapping part of the NI memory space into its own virtual memory; following that, it uses unprivileged load/store instructions

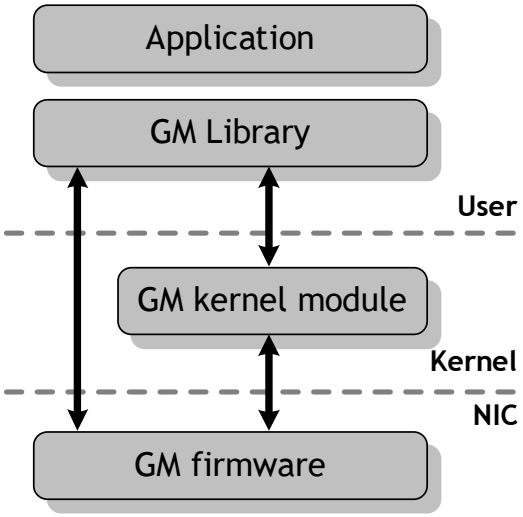


Figure 2.3: *Myrinet/GM stack*

into the relevant VM segments in order to communicate.

The Myrinet NICs reside on the peripheral bus of a cluster node, PCI/PCI-X in the case of Myrinet-2000 used on our testbed. They feature a RISC microprocessor, called the Lanai, which undertakes almost all network protocol processing, a small amount (2MBs) of SRAM for use by the Lanai and three different DMA engines; one is responsible for DMA transfers of message data between host memory and Lanai SRAM over the half-duplex PCI/PCI-X bus, while the other two undertake transferring data between Lanai SRAM and the full-duplex 2+2Gbps fiber link. To provide user level networking facilities to applications, the GM message-passing system is used [Myr03]. GM comprises the firmware executing on the Lanai, an OS kernel module and a userspace library. These three parts coordinate in order to allow direct access to the NIC from userspace, without the need to enter the kernel via system calls (OS bypass) while maintaining system integrity and ensuring process isolation and memory protection.

In Fig. 2.4 the main components onboard a Myrinet M3F-PCI64B-2 NIC are displayed, i.e., the DMA engines (one on the PCIDMA chip and two on the packet interface), the Lanai and its SRAM. This older version of the Myrinet-2000 NIC uses distinct chips and is displayed for clarity. Our testbed is based the on latest version of the NIC, the M3F2-PCIXE-2. It is built around a newer version of the Lanai, the Lanai2XP, which integrates all of the described functionality in a single chip and supports two packet interfaces.

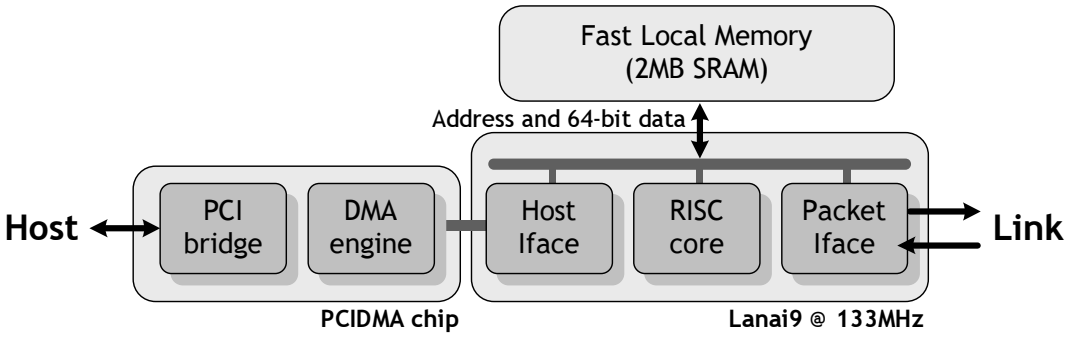


Figure 2.4: *Main components onboard a Myrinet NIC*

Zero-copy user level communication is accomplished by mapping parts of Lanai SRAM (called *GM ports*) into the VM address space of an application. This is a privileged operation, which is done via system calls to the GM kernel module during the application's initialization phase. Each port acts as a communication endpoint for the application. Each port has an unprotected part, which is mapped to userspace and contains queues of send and receive descriptors to be manipulated directly by the application. There is also a protected, trusted part which contains internal port state information and is only accessible by the kernel module and firmware.

The GM firmware polls the port queues periodically, in order to detect any newly posted request. In case of a send operation, it uses DMA in order to transfer the required data from host RAM to Lanai SRAM, then from Lanai SRAM to the NIC of the receiver node, while the reverse happens during a receive.

GM offers reliable, connectionless point-to-point message delivery between different ports, by multiplexing message data from multiple ports over connections kept from each host to every other host in the network. A "Go back N" protocol is used, trading bandwidth for reduced latency and software overhead.

The GM firmware (or Myrinet Control Program) is organized in four state machines, called SDMA, SEND, RECV and RDMA, with each state machine being responsible for a specific part of protocol processing.

The SDMA state machine polls all open ports for new send events posted by the application, creates the relevant send tokens and inserts them in the appropriate connection queue, based on the target node. It also notices when an application posts a new buffer for incoming messages and creates the corresponding receive token. The SDMA engine initiates read DMA transactions to transfer packet data ("chunks") from host memory

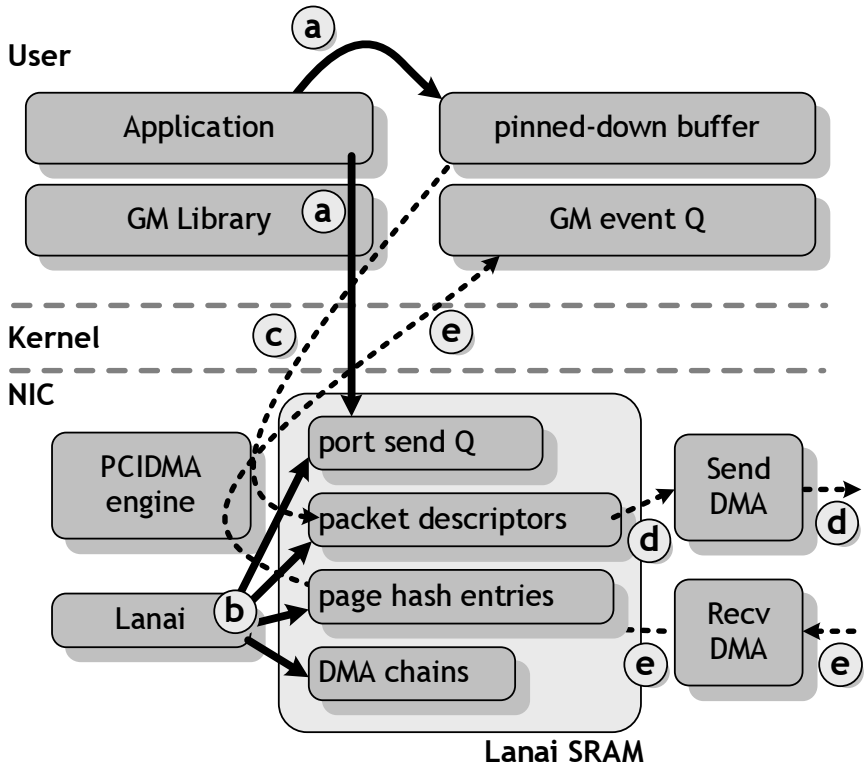


Figure 2.5: *Sending a message over Myrinet/GM*

into packet buffers in Lanai SRAM and enqueues the packets to the SEND state machine for injection into the network. To ensure reliable packet delivery, an associated "send record" is created whenever a new packet is enqueued. It contains information on the origin of the packet payload inside the message buffer and the sequence number of the packet, to be acknowledged by the receiver later on.

The SEND state machine receives packets prepared by the SDMA machine and any pending (N)ACK packets prepared by the RDMA machine and injects them into the network by programming the Send DMA engine of the packet interface.

Incoming packets from the wire are received by the RECV state machine. Those containing data for incoming messages are passed to the RDMA state machine, while control packets are handled by the RECV machine, manipulating the relevant send records directly. For an acknowledgment packet, the relevant send records are freed and the sequence number of the last ACKed packet is recorded. If the last packet for a pending send token is completed, the token is freed and a "send complete" notification event is passed to the userspace application. For a negative acknowledgment packet the send

records are freed but the connection is also rewound so that data are resent, by restoring message send pointers to the values stored in the send records (the "Go back N" property of the communications protocol).

The RDMA state machine accepts incoming data packets from the RECV state machine and tries to match them with a pre-posted application buffer (a pending receive token) based on its tag ("size" in GM parlance). If it succeeds, it initiates a write DMA transaction to transfer the chunk into host memory. When all of the message has been received into the buffer, the application is notified by DMAing an event record into its GM event queue. The RDMA state machine is also responsible for generating (N)ACK control packets, by comparing the sequence numbers of incoming packets for each connection with their expected values.

As with any other user level networking architecture, the design of GM needs to address the issues of data movement, address translation, protection and completion notification described in the previous section. To understand how GM fits in this design space, we present the basic steps that need to take place for a standard GM send operation to complete successfully, as displayed in Fig. 2.5.

In all similar figures, the solid lines lines correspond to Programmable I/O (PIO), involving either the CPU or the Lanai. The dashed lines correspond to DMA operations.

The basic GM send primitive `gm_send_with_callback()` entails the following steps, which correspond to **(a)**-**(e)** in Fig. 2.5.

(a) Buffer allocation, creation of send event The application computes the message to be sent in a pinned-down userspace buffer. In GM all message data movement happens using DMA, without any coordination with the kernel. Thus, to ensure that a GM userspace buffer remains in memory and is not swapped out to disk, the relevant memory pages must be *registered* with the kernel. Allocation of DMAable memory cannot be completed in userspace; GM-specific memory registration calls to the GM kernel module are used beforehand, so that the buffer resides in DMA-able host memory is never swapped out by the kernel's VM subsystem. Afterwards, the application uses the GM user library to construct and place a "send event" structure that contains the virtual address of the buffer to be transferred in the send queue of an open port, in mapped Lanai SRAM.

(b) Address translation and Send DMA The SDMA state machine of the GM firmware notices the send event as it polls the send queues of open ports periodically. It constructs the corresponding send token and places it in the queue of pending send tokens. From there, it is picked up when the PCIDMA engine becomes idle and a new chunk can be prepared for injection in the network. The SDMA engine needs to program the PCIDMA engine using physical addresses in order to fetch packet message data into SRAM, so it needs to perform virtual-to-physical address translation. To maintain process isolation and memory protection, GM keeps track of virtual to physical mappings in private, GM-specific pagetables residing in kernel host memory. They are updated at page registration time by the privileged kernel module. To mitigate the cost of translation, the firmware keeps a cache in Lanai SRAM (the "page hash entries" region) and fetches the needed translations from the host-based pagetables using DMA in case of a cache miss. If no translation is found, even after consulting the host page tables, bogus data are fetched from a specially allocated page in host RAM. The SDMA engine allocates a new Myrinet packet descriptor structure, fills the header field and programs the PCIDMA engine to fetch the packet payload from host RAM. It also creates a send record for the packet.

(c) Transfer of payload using DMA Message data are brought via DMA into the Lanai SRAM. The packet descriptors are queued as pending for transmission over the wire and passed to the SEND state machine.

(d) Injection of packets into the network The SEND state machine programs the Send DMA engine of the packet interface to retrieve data from Lanai SRAM and put it on the link.

(e) Acknowledgment by the remote side An ACK packet is received by the remote side and picked up by the RECV state machine. The relevant send record is freed. Assuming it was the last for the message, the firmware needs to notify the application by placing an appropriate "receive complete" event in the application's event queue, residing in host memory, using DMA. The application takes notice of the event by polling the queue periodically. Alternatively, it can enter the kernel, requesting to be woken up when a new event arrives. In this case, the firmware will interrupt the host, so that the GM module unblocks it.

The process is similar in the case of receiving a message. In this case, the RDMA state machine takes over, matches the incoming data with a posted message buffer and generates ACK packets to be transmitted back to the sender.

It is important to note that sending – and receiving – a message using GM is in fact a two-phase process:

Host to Lanai DMA: Virtual-to-physical translation takes place, the PCIDMA engine starts, message data are copied from host RAM to Lanai SRAM

Lanai to wire DMA: Message data are retrieved from SRAM and sent to the remote NIC by the Send DMA engine.

2.3 The Linux Block Layer

This section provides a short description of the Linux block layer. We are concerned mainly with its functionality and core data structures, in order to gain a better understanding of the context in which a block device driver operates. As can be seen in Fig. 2.2, the client-side portion of an nbd system commonly resides in kernelspace, implementing a virtual block device driver for use by the rest of the system. Thus, a short discussion of the interaction between an OS kernel and block device drivers is necessary.

The Linux block layer is a representative example of a modern, production-quality I/O infrastructure supporting a variety of I/O optimizations, such as request coalescing, request scheduling, and scatter-gather DMA.

Although the various implementation details are Linux-specific, the basic concepts remain constant for all modern Operating Systems. The following discussion is valid as of kernel 2.6.30.

2.3.1 Main functionality

The Linux block layer lies between kernel code which needs to perform block I/O, e.g., filesystems, and the actual block device drivers (Fig. 2.6). For reasons of efficiency, requests are not handled by device drivers directly. They are placed in queues inside the

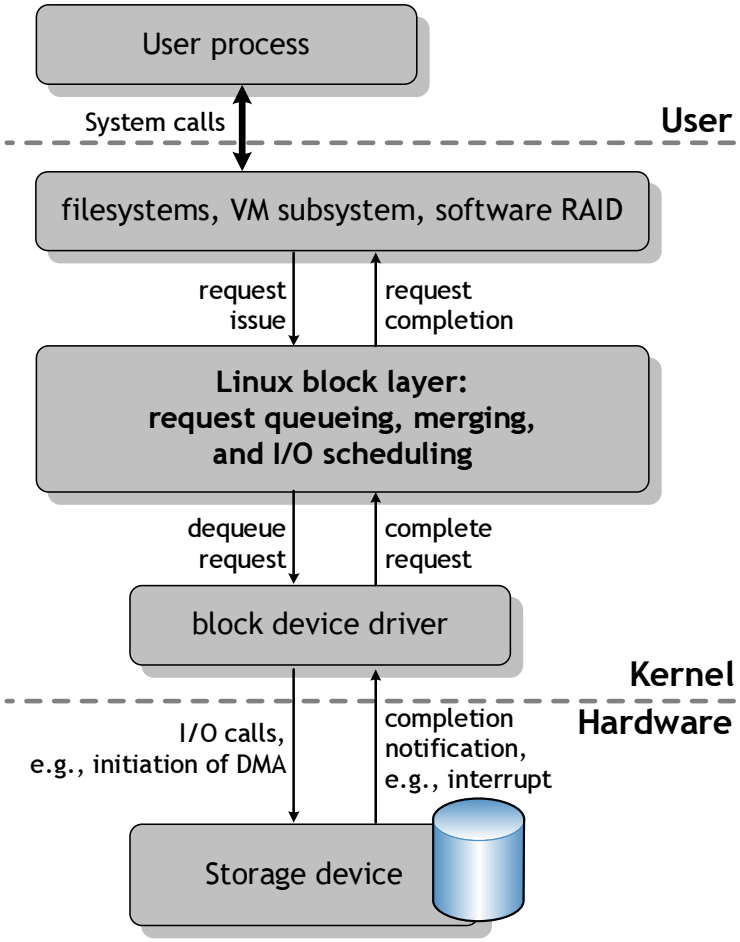


Figure 2.6: *Position of the block layer inside the Linux kernel*

Linux block layer, until they are extracted by block device drivers for processing. The block layer aims to transform requests while in the queue, to improve performance; it may merge incoming requests for consecutive sectors and coalesce references to neighboring memory segments. These requests can then be presented as a single unit of work to the device driver and underlying device.

The block layer may re-order requests before presenting them to the driver, a process known as *I/O scheduling*. This is done to minimize disk seek operations, since they dominate service time for spindle-based storage, or to enforce I/O prioritization among processes. The block layer supports a plug-in architecture, which enables per-queue setting of the scheduling algorithm dynamically. A number of different I/O schedulers is provided with the Linux kernel, namely the *no-op*, *CFQ*, *deadline* and *anticipatory* [ID01] schedulers.

Finally, the block layer provides an interface for device drivers to report *completion* of

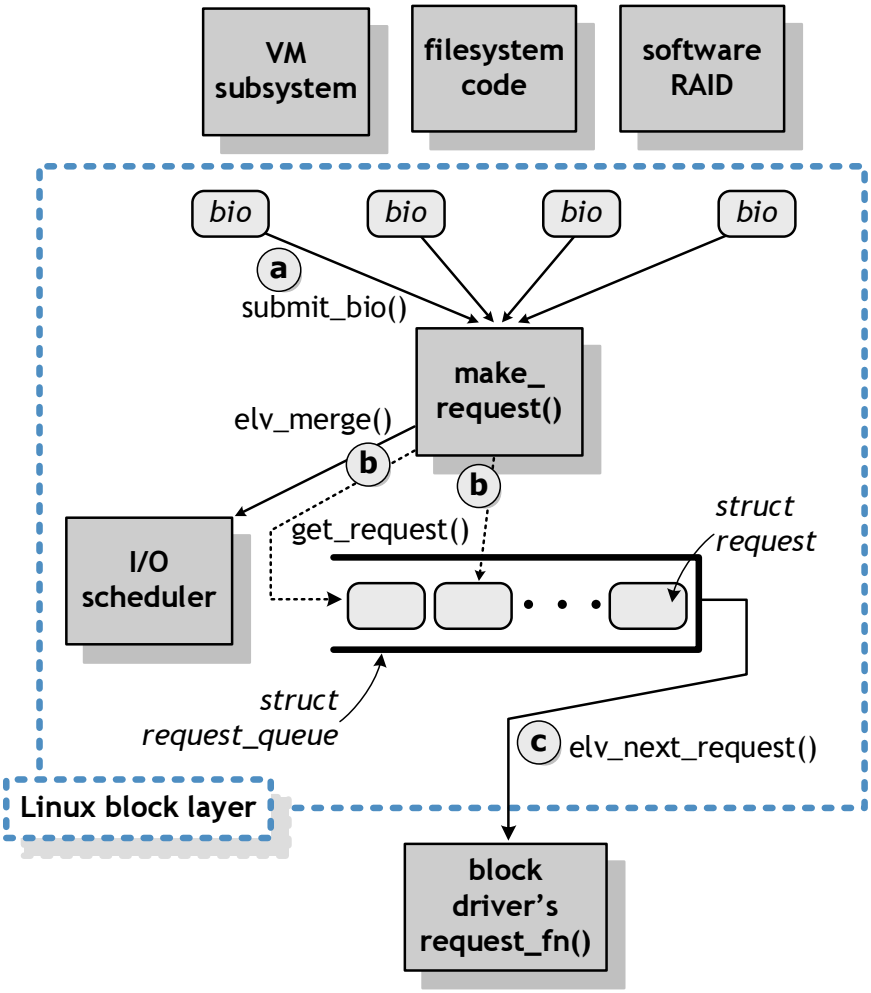


Figure 2.7: *The Linux block layer servicing an I/O request*

I/O operations, when the requested data transfer operation has been carried out by the storage device. These completion events are then propagated to the originators of the requests, and may trigger state changes in userspace processes.

Thus, the functionality of the block layer comprises request submission, queuing, merging, memory segment coalescing, I/O scheduling and I/O operation completion. In the following, we refer briefly to each step of I/O request processing, by following the course of an I/O request from creation to completion (Fig. 2.7). We also provide a description of the most important data structures implicated at every step.

2.3.2 Servicing a block I/O request

Request submission - The `bio` structure

The `bio` structure is the core data structure of the Linux block I/O layer. It represents a block request from the point of view of a kernel entity doing disk I/O. Such entities include filesystem code traversing on-disk structures, the VM subsystem issuing page fill and page writeback requests to swap pages in and out of the page cache, or a software RAID device servicing requests by distributing them to underlying storage devices.

The `bio` is the basic unit of I/O submission; whenever a part of the kernel needs to move data from and to secondary storage, it allocates a `bio` structure, fills it properly, then submits it to the block layer for further processing using the `submit_bio()` interface (Fig. 2.7, step (**a**)). Similarly, the `bio` is the basic unit of I/O completion; each `bio` may define a completion callback function, which is called upon completion of the related block transfer. Thus, the originator of the `bio` has a chance to perform any necessary bookkeeping as `bio` completion is reported by the block layer, e.g., awaking blocked userspace processes.

A `bio` defines a block I/O operation as a group of consecutive sectors that need to be read or written to dispersed memory segments. For this discussion, the fields which are of interest are:

- `unsigned long bi_flags`: A set of bits describing various `bio` attributes, among them the direction of data transfer, whether the `bio` reads or writes data.
- `sector_t bi_sector`: The first 512-byte sector to be transferred for this request.
- `unsigned int bi_size`: The amount of data to be transferred in bytes. The macro `bio_sectors(bio)` is used to convert this into sectors.
- `struct block_device *bi_bdev`: The block device this `bio` will be serviced from. A stacking block driver, e.g., software RAID, may modify this field to redirect this `bio` for service by a different device.
- `unsigned short bi_phys_segments`: The number of disjoint physical memory segments referenced by this `bio`.

- `struct bio_vec *bi_io_vec`: Pointer to an array of vectors into physical memory. Each vector of type `bio_vec` describes a contiguous area in physical memory as the tuple {page of physical memory, length in bytes, offset into page}. The physical page is defined as a pointer to the associated page frame, the `struct page` used by the kernel for managing a single page of physical memory [Gor04].

- `bio_end_io_t *bi_end_io`: Pointer to a callback function, invoked by the block layer upon completion of this `bio`, in interrupt context.

Essentially, the `bi_io_vec` array implements a scatter-gather list. Note that it points to physical memory locations directly; The kernel may have to establish mappings in its virtual memory space, before code on the CPU can refer to the actual data.

Request queueing and merging, I/O scheduling

At the bottom end of the processing hierarchy, lie the block device drivers and the actual hardware devices. Block device drivers do not generally receive `bio` structures directly; they service block requests received as `struct request`s, which emerge after the `bio` structures have undergone processing by the Linux block layer.

When a new `bio` arrives, the block layer first tries to merge it with a pre-existing request, or creates a new `struct request` for it and inserts it in the queue for the corresponding block device (Fig. 2.7, step **(b)**, function `make_request()`). To determine whether a `bio` is eligible for merging (i.e., it refers to locations adjacent to those of a pre-existing request), the block layer calls into the I/O scheduler (`elv_merge()`). This allows the scheduler to enforce further scheduler-specific constraints.

While organizing `bio`s in `struct request`s, the kernel maintains a number of restrictions: *(a)* The request must continue to refer to a contiguous area on the storage device, *(b)* All `bio` structures must be of the same direction, describing either read or write operations, *(c)* The request must not exceed hardware limits imposed by the underlying block device, such as the maximum length of DMA segments supported.

To satisfy the third requirement, the block layer offers a rich set of functions to device drivers, through which they may fine-tune the process of request merging and scheduling by setting hardware and driver-imposed limits. These limits include the maximum

number of sectors which may be included in a single request, the maximum number and length of disjoint memory segments supported, and the maximum physical address that the hardware device may perform DMA with. These tunables are maintained as per-queue attributes and are consulted by the block layer before allowing a merge operation.

I/O requests are queued by the block layer and dequeued by device drivers to be serviced. To allow a number of sufficiently large requests to be built before the driver has a chance to retrieve and pass them to the hardware, the block layer supports *queue plugging*; when a queue is plugged, the I/O scheduler is actively adding and merging requests in the queue. When the queue becomes unplugged, because it has remained plugged for a predetermined, tunable duration, or the number of requests exceed a certain threshold, the queue switches to the unplugged state. At this time, the block layer activates the driver, by calling the *request function* associated with the queue.

Request extraction, servicing and completion

The driver's request function is responsible for extracting the next request from the queue and passing it to the storage device for servicing, until the queue becomes empty or the maximum number of outstanding requests is reached. To retrieve the next request, the request function calls into the scheduler (Fig. 2.7, step (**c**), `elv_next_request()`). Servicing a request commonly entails initiating a DMA transaction for the associated memory segments.

When the driver is notified that all of the data have been transferred, either via polling the device or via an interrupt, it needs to *complete* the request with the block layer, usually by calling `end_request()` for the related `struct request`. When a request completes, its associated `bio`s are completed; if a callback function has been defined for this `bio`, it gets to run at this point. Thus, completion notification reaches all the way up to the originators of the individual requests.

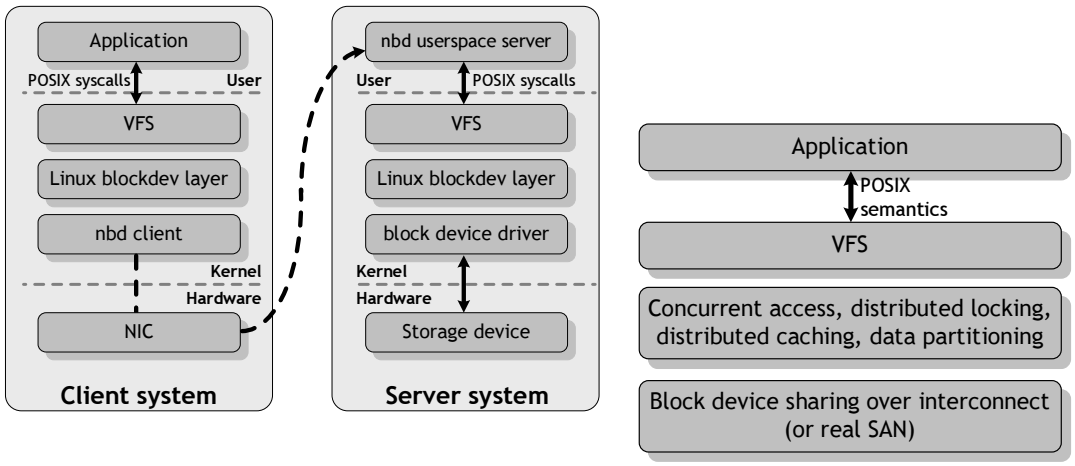
3 Design and implementation of gmblock

This section presents the design and implementation of the gmblock nbd system over Myrinet/GM. We begin by discussing the overheads involved in standard TCP/IP and RDMA-based approaches, and how gmblock's design evolves from these. Then, we present the necessary changes to GM and the Linux kernel in order to support a prototype implementation. Experimental evaluation of the proposed path shows significant improvement in sustained throughput and reduced interference on the host's memory bus. The chapter concludes with a discussion of various aspects of gmblock's design, namely its ability to serve structured data, the effect on server and client-side caching and prefetching, proposed architectural changes to support similar functionality with interconnects other than Myrinet, and its applicability to low-frequency, low-power embedded storage servers.

3.1 Design of gmblock's nbd mechanism

3.1.1 Traditional nbd designs

The main principle behind an nbd client-server implementation is portrayed in Fig. 3.1(a). The nbd client usually resides in the OS kernel and exposes a block device interface to the rest of the kernel, so that it may appear as an ordinary, directly-attached storage device. The requests being received from the kernel block device layer are encapsulated



(a) Generic nbd system (b) Parallel fs on top of nbd

Figure 3.1: *A parallel filesystem executing over an nbd infrastructure*

in network messages and passed to a remote server. This server is commonly not run in privileged kernelspace. Instead, it executes as a userspace process, using standard I/O calls to exchange data with the actual block device being shared.

The pseudocode for a generic nbd server can be seen in Fig. 3.2. For simplicity, we will refer to a remote block read operation but the following discussion applies to write operations as well, if the steps involving disk I/O and network I/O are reversed. There are four basic steps involved in servicing a read block request: **(a)** The server receives the request over the interconnect, unpacks it and determines its type – let's assume it's a read request **(b)** A system call such as `lseek()` is used to locate the relevant block(s) on the storage medium **(c)** The data are transferred from the disk to a userspace buffer **(d)** The data are transmitted to the node that requested them.

The overhead involved in these operations depends significantly on the type of interconnect and the semantics of its API. To better understand the path followed by the data at the server side, we can see the behavior of a TCP/IP-based server at the logical layer, as presented in Fig. 3.3(a). Again, solid lines denote PIO operations, dashed lines denote DMA operations. **(a)** As soon as a new request is received, e.g., in the form of Ethernet frames, it is usually DMAed to kernel memory, by the NIC **(b)** Depending on the quality of the TCP/IP implementation it may or may not be copied to other buffers, until it is copied from the kernel to a buffer in userspace. The server process, which presumably blocks in a `read()` system call on a TCP/IP socket, is then woken up to

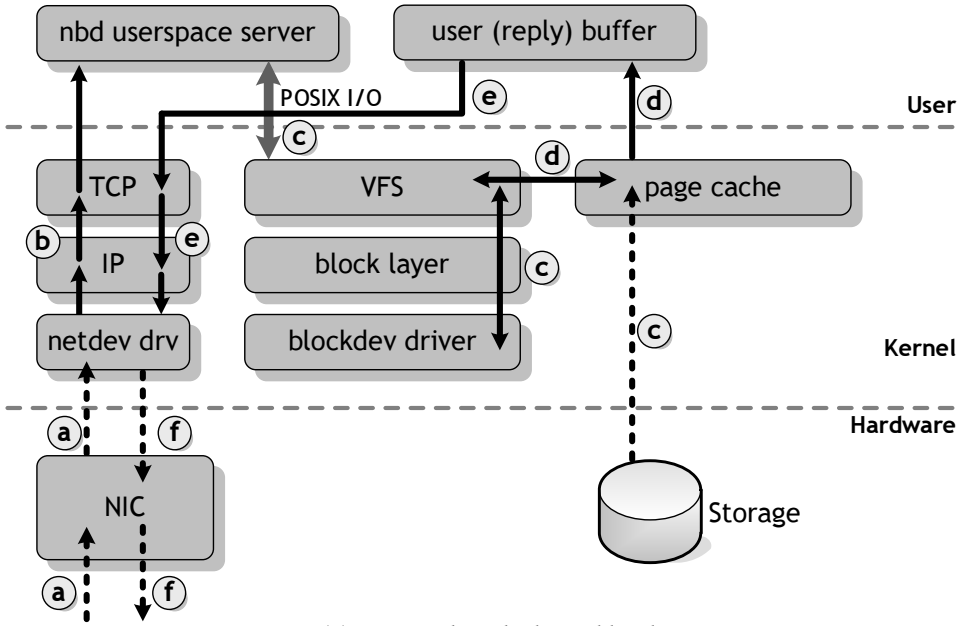
```
initialize_interconnect();
fd = open_block_device();
reply = allocate_memory_buffer();
for (;;) {
      cmd = recv_cmd_from_interconnect();
      lseek(fd, cmd->start, SEEK_SET);
      switch (cmd->type) {
         case READ_BLOCK:
            read(fd, &reply->payload, cmd->len);
         case WRITE_BLOCK:
            write(fd, &req->payload, cmd->len);
      }
      insert_packet_headers(&reply, cmd);
      send_over_net(reply, reply->len);
}
```

Figure 3.2: *Pseudocode for an nbd server*

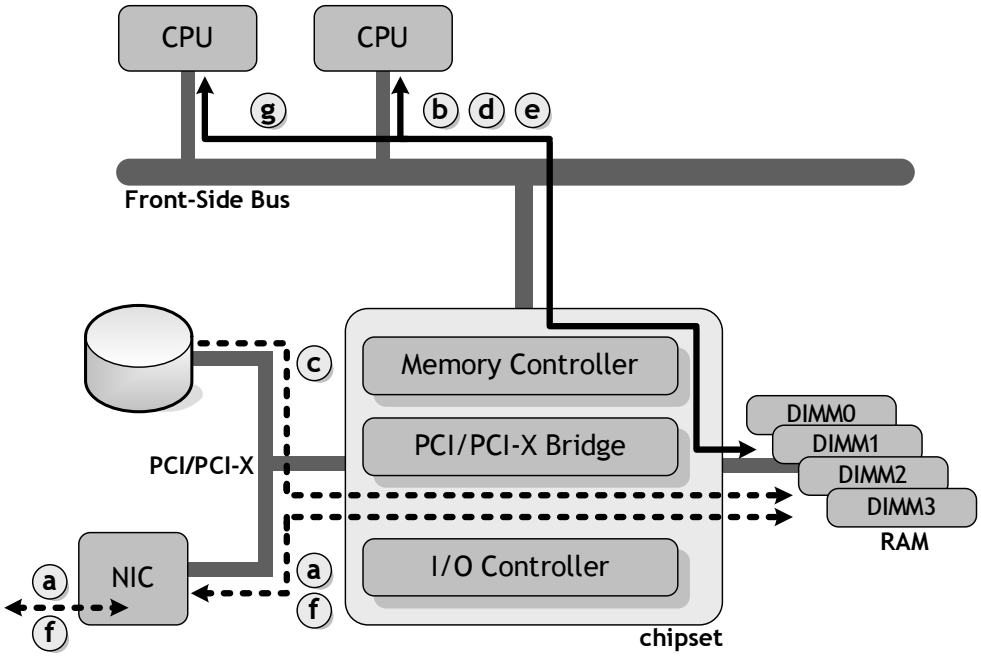
process the request. It processes the request by issuing an appropriate `read()` call to a file descriptor acquired by having `open()`ed a block device. In the generic case this is a *cached* read request; the process enters the kernel, which (**c**) uses the block device driver to setup a DMA transfer of block data from the disk(s) to the *page cache* kept in kernel memory (**d**) Then, the data need to be copied to the userspace buffer and the `read()` call returns (**e**) Finally, the server process issues a `write()` call, which copies the data back from the userspace buffer into kernel memory. Then, again depending on the quality of the TCP/IP implementation, a number of copies may be needed to split the data in frames of appropriate size, which are (**f**) DMAed by the NIC and transferred to the remote host.

In Fig. 3.3(b), we can see the actual path followed by data, at the physical level. The labels correspond one-to-one with those used in the previous description: (**a, b**) Initially, the read request is DMAed by the NIC to host RAM, then copied to the userspace buffer using PIO (**c**) The disk is programmed to DMA the needed data to host RAM. The data cross the peripheral bus and the memory bus (**d**) The data are copied from the page cache to the userspace buffer by the CPU (**e**) The data are copied back from the userspace buffer to the kernel, to be sent over the network (**f**) The data cross the memory bus and the peripheral bus once again, to be sent over the network via DMA.

This data path involves a lot of redundant data movement. The number of copies needed to move data from the disk to the TCP/IP socket can be reduced by allowing the kernel



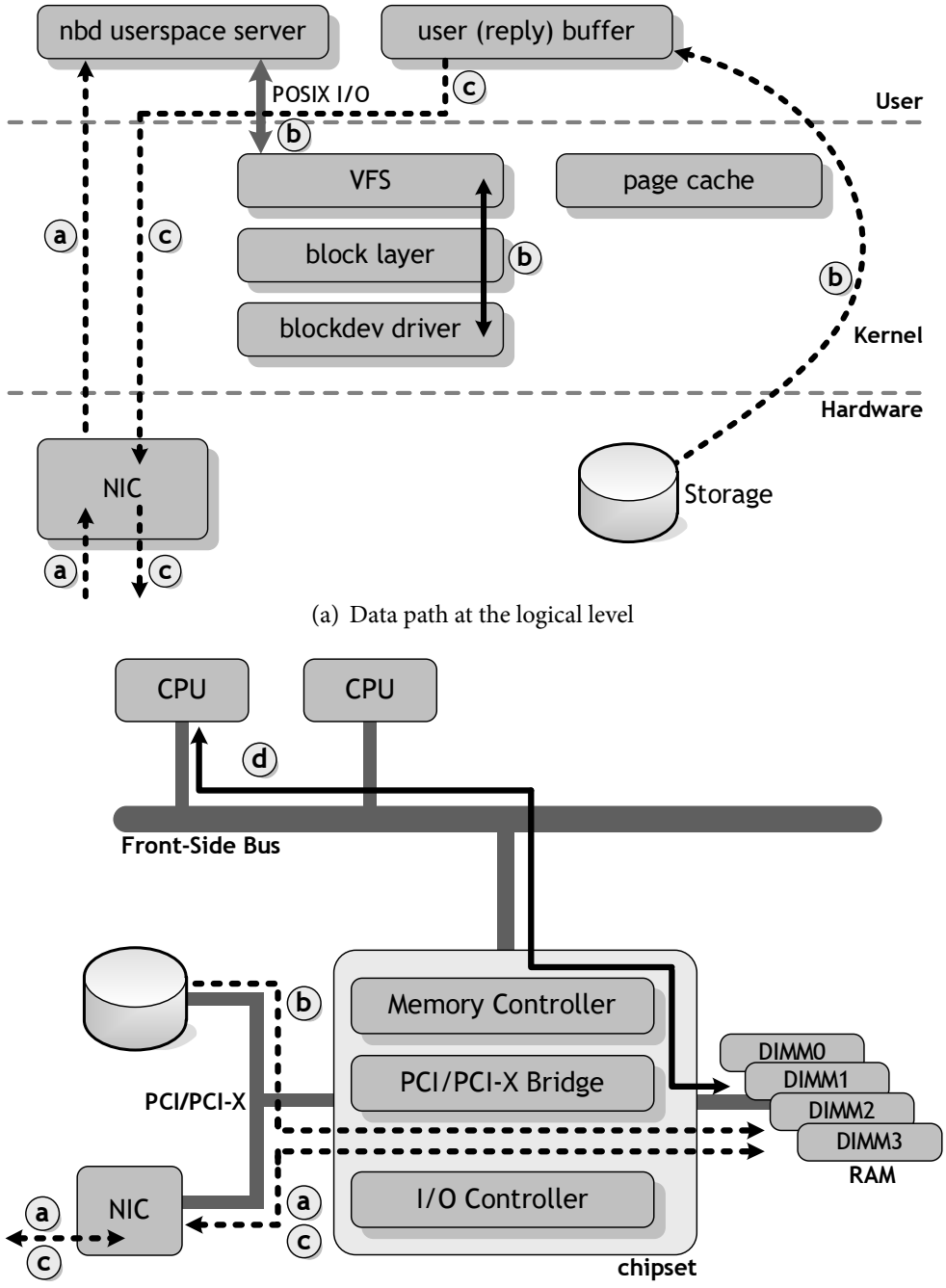
(a) Data path at the logical level



(b) Data path at the physical level

Figure 3.3: *TCP/IP based nbd server*

more insight into the semantics of the data transfer; one could map the block device onto userspace memory, using `mmap()`, so that a `write()` to the socket copies data directly from the page cache to the network frame buffers, inside the kernel. However, depending on the size of the process address space, not all of the block device may be mappable. Thus, the overhead of remapping different parts of the block device must be taken into account.



(a) Data path at the logical level

(b) Data path at the physical level

Figure 3.4: *GM-based nbd server*

A different way to eliminate one memory copy is by bypassing the page cache altogether. This can be accomplished by use of the POSIX `O_DIRECT` facility, which ensures that all I/O with a file descriptor bypasses the page cache and that data are copied directly into userspace buffers. The Linux kernel supports `O_DIRECT` transfers of data; the block layer provides a generic `O_DIRECT` implementation which takes care of pinning down the relevant userspace buffers, determining the physical addresses of the pages involved

and finally enqueuing block I/O requests to the block device driver which refer to these pages directly instead of the page cache. Thus, if the block device is DMA-capable, the data can be brought into the buffers directly, eliminating one memory copy. Still, they have to be copied back into the kernel when the TCP/IP `write()` call is issued.

The main drawback of this data path is the large amount of redundant data copying involved. If only one kernel copy takes place, data cross the peripheral bus twice and the memory bus four times – one per DMA transfer and twice for the CPU-based memory copy. When forming virtual storage pools by having compute nodes export storage to the network, remote I/O means fewer CPU cycles and less memory bandwidth are available to the locally executing workload (path (**g**)). Moreover, doing CPU-based memory movement leads to cache pollution, evicting parts of the working set of the local workload.

The problem is alleviated, if a user level networking approach is used. When a cluster interconnect such as Myrinet is available, the OS kernel can be bypassed during the network I/O phase, by extending the nbd server application so that GM is used instead of TCP/IP. In this case, some of the redundant copying is eliminated, since the steps to service a request are (Fig. 3.4(a)): (**a**) A request is received by the Myrinet NIC and copied directly into a pinned-down request buffer (**b**) The server application uses `O_DIRECT`-based I/O so that the storage device is programmed to place block data into userspace buffers via DMA (**c**) The response is pushed to the remote node using `gm_send()`, as in Section 2.2.2. In this approach, most of the PIO-based data movement is eliminated. The CPU is no longer involved in network processing, the complex TCP/IP stack is removed from the critical path and almost all CPU time is devoted to running the computational workload. However, even when using GM for message passing, main memory is still on the critical path. At the physical layer (Fig. 3.4(b)), for a read operation, block data are transferred from the storage devices to in-RAM buffers, then from them to the Myrinet NIC. Thus, they traverse the peripheral bus and the main memory bus twice; pressure on the peripheral and main memory buses remains, and remote I/O still interferes with local computation (**d**), since they contend for access to RAM.

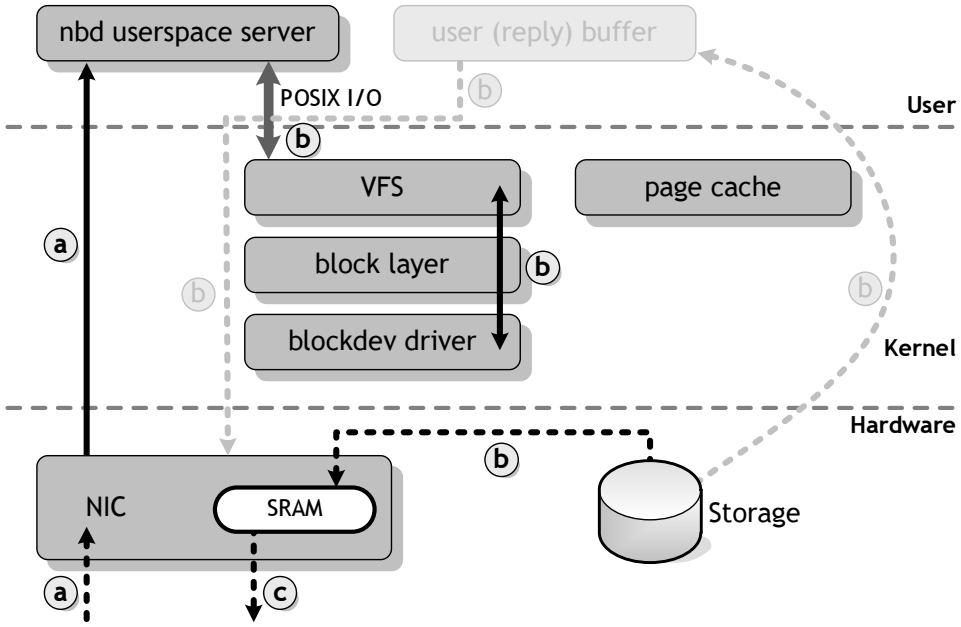
3.1.2 GMBlock: An alternative data path with memory bypass

To solve the problem of redundant data movement at the server side of an nbd system, we propose a shorter data path, which does not involve main memory at all. To service a remote I/O request, all that is really needed is to transfer data from secondary storage to the network or vice-versa. Data flow based on this alternative data path is presented in Fig. 3.5(b): **(a)** A read request is received by the Myrinet NIC **(b)** The nbd server process services the request by arranging for block data to be transferred directly from the storage device to the Myrinet NIC **(c)** The data is transmitted to the node that initiated the operation.

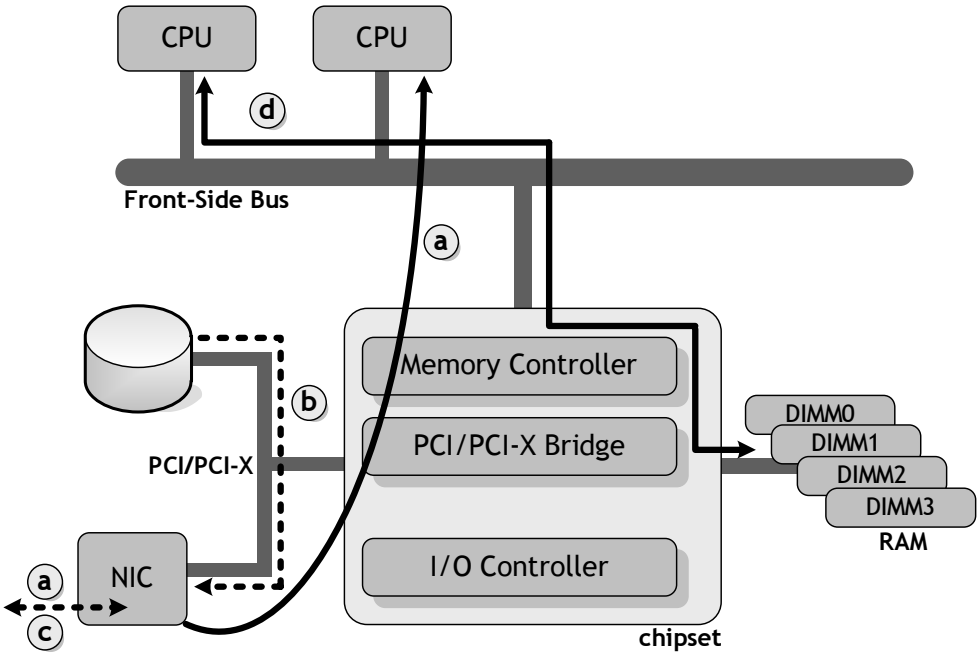
Implementing this path would solve most of the problems described above:

- The critical path is the shortest possible. Data go directly from disk to NIC or vice-versa.
- The full capacity of the peripheral bus can be used, since data only traverse it once.
- There is no staging in buffers kept in RAM, thus no memory bandwidth is consumed by I/O and code executing on local CPUs does not incur the overhead of memory contention.

Most importantly, this design would acknowledge the fact that the remote I/O path may be disjoint from main memory. The inclusion of RAM buffers in all previous data paths is a necessity arising from the programming semantics of the mechanisms used to enable the transfer – GM and Linux kernel drivers – rather than from the intrinsic properties of remote I/O operations; GM programs the DMA engines on the Myrinet NIC to exchange data between the Lanai SRAM and RAM buffers, while the kernel programs storage devices to move data from/to page cache or userspace buffers kept in main memory. Thus, to support the proposed data path, we need to extend these mechanisms so that direct disk-to-NIC transfers are supported. At the same time, the architecture-dependent details of setting up such transfers must be hidden behind existing programming abstractions, i.e., GM user level networking primitives and the Linux I/O system call interface. In this approach, only minimal changes to the the nbd server source code will be required to support the enhanced functionality.



(a) Data path at the logical level



(b) Data path at the physical level

Figure 3.5: *Proposed gmblock server*

Let's assume a GM-based nbd server servicing a read request similar to that of Fig. 3.2. In the case of GM, the server would have used `gm_open()` to initialize the interconnect, and `gm_dma_malloc()` to allocate space for the message buffer. Variable `reply` contains the virtual address of this buffer, dedicated to holding the reply of a remote read operation, before it is transferred over Myrinet/GM. If this memory space was not allocated in RAM, but could be made to reside in Lanai SRAM instead, then the `read()` system

call could be used un-altered, to express the desired semantics; It would still mean "I need certain blocks to be copied to memory pointed to by `reply`", this time however referring to a buffer in SRAM, mapped onto the process's VM space at location `reply`.

However, if standard, buffered I/O was used, using this call would first bring the data into the kernel's page cache, then a CPU-based `memcpy()` would be used to copy the data from the cached page to the mapped SRAM buffer. This chain would still invoke PIO; the whole of Lanai SRAM is exposed as a large memory-mapped I/O resource on the PCI physical address space. Thus, every reference by the CPU to the virtual address space pointed to by `reply` during the `memcpy()` operation, would lead to I/O transactions over the peripheral bus. The situation would be radically different, if POSIX `O_DIRECT` access to the open file descriptor for the block device was used instead. In this case, the kernel would bypass the page cache. Its direct I/O subsystem would translate the virtual address of the buffer to a physical address in the Myrinet NIC's memory-mapped I/O space and use *this* address to submit a block I/O request to the appropriate in-kernel driver. In the case of a DMA-capable storage device, the ensuing DMA transaction would have the DMA engine copying data directly to the Myrinet NIC, bypassing the CPU and main memory altogether. To finish servicing the request, the second half of a GM Send operation is needed: the Host-to-Lanai DMA phase is omitted and a Lanai-to-wire DMA operation is performed to send the data off the SRAM buffer.

Conversely, in the case of a remote write operation, the DMA-capable storage device would be programmed to retrieve incoming data directly from the Lanai SRAM buffer after a wire-to-Lanai DMA operation completes.

It is important to note that almost no source code changes are needed in the nbd server to support this enhanced data path. The server process still issues `read()` or `write()` and `gm_send()` calls, unaware of the underlying transfer mechanism. The desired semantics emerge from the way the Linux block driver layer, the kernel's VM subsystem and GM's user level networking capabilities are combined to construct the data path.

3.2 Implementation details

The implementation of gmblock's optimized data path involves changing two different subsystems: First, the user level networking infrastructure provided by GM must be

extended to support GM buffers in Lanai SRAM. Second, the Linux VM mechanism must include support for treating Lanai SRAM as host RAM, so that direct I/O from and to PCI memory-mapped I/O regions is possible.

3.2.1 GM support for buffers in Lanai SRAM

The GM middleware needs to be enhanced, as to allow the allocation, mapping and manipulation of buffers residing in Lanai SRAM by userspace applications. At the same time, it is important to preserve GM semantics and UNIX security semantics regarding process isolation and memory protection, as is done for message passing from and to userspace buffers in host RAM.

The described changes were tested on various combinations of GM-2.0 and GM-2.1 on an Intel i386 and an Intel EM64T system. However, the changes affect the platform-independent part of GM, so they should be usable on every architecture/OS combination that GM has been ported to.

This is the functionality that needs to be supported, along with the parts of GM that are affected:

- Allocation of buffers in Lanai SRAM (GM firmware)
- Mapping of buffers onto the VM of a process in userspace (GM library, GM kernel module)
- Sending and receiving messages from/to Lanai SRAM using `gm_send()` and `gm_provide_receive_buffer()` (GM library, GM firmware)

For the first part, the firmware initialization procedure was modified, so that a large, page-aligned buffer is reserved for gmblock's use (Fig. 3.6. The buffer is allocated off the firmware heap, between the growing GM connection array and the GM ports. Our testbed uses Myrinet NICs with 2MB of SRAM, out of which we were able to allocate at most 700KB for gmblock's use and still have the firmware fit in the available space and execute correctly.

The second part involved changes in the GM library, which tries to map the shared memory buffer. The GM kernel module verifies that the request does not compromise

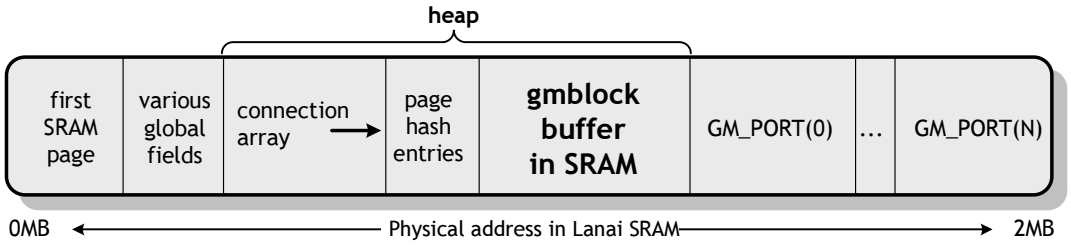


Figure 3.6: *Lanai memory map after allocation of gmblock's SRAM buffer*

system security, then performs the needed mapping. The Lanai SRAM buffer is shared among processes, but different policies may be easily implemented, by changing the relevant code in the GM kernel module.

Finally, to complete the integration of the SRAM buffer in the VM infrastructure and allow it to be used transparently for GM messaging, we enhance the GM library so that the requirement for all message exchange to be done from/to in-RAM buffers is removed. At the userspace side the library detects that a GM operation to send a message or to provide a receive buffer refers to Lanai SRAM, and marks it appropriately in the event passed to the Lanai. There, depending on the type of the request:

- For a send request, the SDMA state machine omits the Host-to-Lanai DMA operation, constructs the needed Myrinet packets and passes them to the SEND state machine directly, without any intermediate copies.

- Things are more complicated when incoming data need to be placed in an SRAM buffer by the RDMA state machine, since incoming packet data are placed in predefined message buffers by the hardware before any message matching can take place. In the common case of receiving into RAM, they are moved to their final destination during the Lanai-to-Host DMA phase. When receiving into SRAM buffers this is replaced by a copy operation, undertaken by a copy engine on the Lanai. The memory arbitration scheme of the LanaiX ensures that the copy progresses without impacting the rate at which concurrent, pipelined packet receives occur.

3.2.2 Linux VM support for direct I/O with PCI ranges

To implement gmblock's enhanced data path, we need to extend the Linux VM mechanism so that PCI memory-mapped I/O regions can take part in direct I/O operations. So far, the GM buffer in Lanai SRAM has been mapped to a process's virtual address space and is accessible using PIO. This mapping translates to physical addresses belonging to the Myrinet NIC's PCI memory-mapped I/O (MMIO) region. The MMIO range lies just below the 4GB mark of the physical address space in the case of the Intel i386 and AMD x86-64 platforms.

To allow the kernel to use the relevant physical address space as main memory transparently, we extend the architecture-specific part of the kernel related to memory initialization so that the kernel builds page management structures (*pageframes*) for the full 4GB physical address range and not just for the amount of available RAM. The relevant `struct page` structures are incorporated in a Linux *memory zone*, called `ZONE_PCIMEM` and are marked as reserved, so that they are never considered for allocation to processes by the kernel's memory allocator.

The proposed modification does not constitute mere *mapping* of physical addresses; rather it concerns the *management* of physical memory by the kernel. With these modifications in place, PCI MMIO ranges are treated by the Linux VM as host RAM. All complexity is hidden behind the page frame abstraction, in the architecture-dependent parts of the kernel; even the direct I/O layer does not need to know about the special nature of these pages.

3.3 Experimental evaluation

To quantify the performance benefits of employing gmblock's short-circuit data path we compare three different nbd systems in a client-server block-level storage sharing configuration. The first one is a prototype implementation of gmblock with message buffers on Lanai SRAM (hereafter `gmblock-sram`) so that direct disk-to-NIC transfers are possible. The second is a standard TCP/IP-based system, Red Hat's GNBD, the reworked version of NBD that accompanies GFS (`tcpip-gnbd`). GNBD v1.03 runs over the same Myrinet, with Ethernet emulation. The third one is gmblock itself, running over GM

	Server A	Server B
Processor	2x Pentium III@1266MHz	Pentium 4@3GHz
Motherboard	Supermicro P3TDE6	Intel SE7210TP1-E
Chipset	Serverworks ServerSet III HE-SL, CIOB20 PCI bridge	Intel E7210 chipset, 6300ESB I/O controller hub
I/O Bus	2-slot 64bit/66MHz PCI	3-slot 64bit/66MHz PCI-X
RAM	2x PC133 512MB SDRAM	2 x PC2700 512MB SDRAM DDR
Disks	8x Western Digital WD2500JS 250GB SATA II	
I/O controller	3Ware 9500S-8 SATA RAID and MBL	
NIC	Myrinet M3F2-PCIXE-2	

Table 3.1: *Hardware specifications of storage servers used in experimental testbed*

without the proposed optimization. Its performance is representative of RDMA-based implementations using a data path which crosses main memory (`gmblock-ram`).

The evaluation concerns three metrics: *(a)* the sustained bandwidth for remote read operations *(b)* the sustained bandwidth for remote write operations and *(c)* the server-side impact on local executing computational workloads. At each point in the evaluation we identify the performance-limiting factor and try to mitigate its effect, in order to observe how the different architectural limitations come into play.

We experiment with storage servers of two different configurations: Server A is an SMP system of two Pentium III processors, with a 2-slot PCI bus, while Server B is a Pentium 4 system, with more capable DDR memory and a 3-slot PCI-X bus. The exact specifications can be found in table 3.1.

The storage medium to be shared over Myrinet is provided by a 3Ware 9500S-8 SATA RAID controller, which has 8 SATA ports on a 64bit/66MHz PCI adapter. We built a hardware RAID0 array out of 8 disks, which distributes data evenly among the disks with a chunk size of 64KB and is exported as a single drive to the host OS. We use two nodes, one of configuration A functioning as the client, the either as the server (of either configuration A or B). The nodes are connected back-to-back with two Myrinet M3F2-PCIXE-2 NICs. The NICs use the Lanai2XP@333MHz processor, with 2MB of SRAM and feature two 2+2Gbit/s full-duplex fiber links. Linux kernel 2.6.22, GM-2.1.26 and 3Ware driver version 2.26.02.008 are used. The I/O scheduler used is the anticipatory scheduler (AS).

We also experiment with a custom solid-state storage device built around another Myrinet

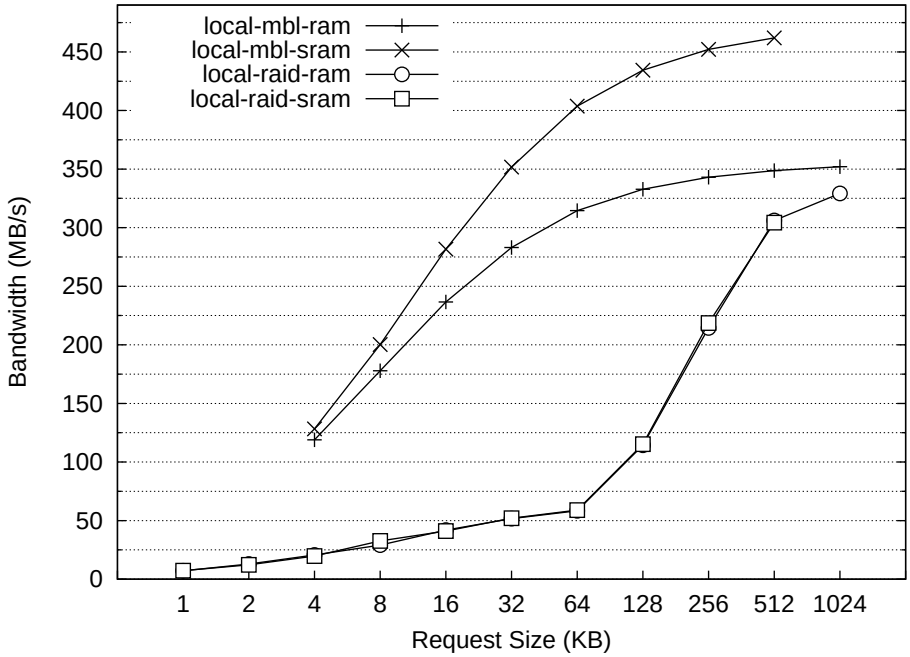
M3F2-PCIXE-2 NIC, which is able to deliver much better throughput even for very small request sizes. We have written a Linux block device driver and custom firmware for the card, which enables it to be used as a standard block device. Block read and write requests are forwarded by the host driver (the Myrinet BLock driver, or "MBL") to the firmware, which programs the DMA engines on the NIC to transfer block data from and to Lanai SRAM. Essentially, we use the card to simulate a very fast, albeit small, storage device which can move data close to the rate of the PCI-X bus.

3.3.1 Experiment 1a: Local disk performance

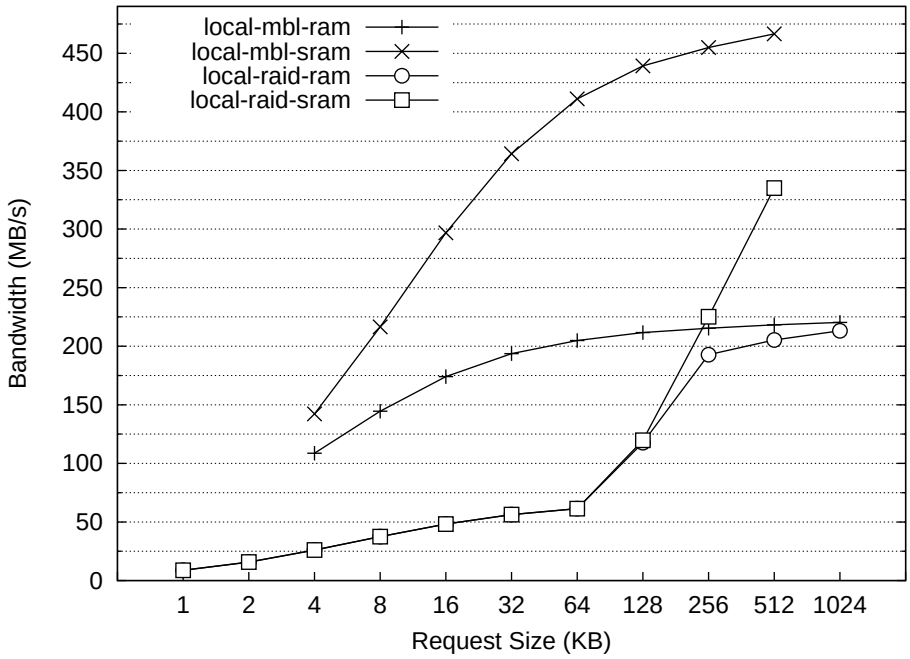
We start by measuring the read bandwidth delivered by the RAID controller, locally, performing back-to-back direct I/O requests of fixed size in the range of 1, 2, . . ., 512KB, 1024KB. The destination buffers reside either in RAM (`local-raid-ram`) or in Lanai SRAM (`local-raid-sram`, short-circuit path) We repeat this experiment for the MBL device as well (`local-mbl-ram`, `local-mbl-sram`). In the following, 1MB = 2^{20} bytes. With the adapters installed on the 64bit/66MHz PCI-X bus of Server B, we get the bandwidth vs. request size curves of Fig 3.7(b).

A number of interesting conclusions can be drawn. First, for a given request size these curves provide an upper bound for the performance of our system. We see that the RAID throughput increases significantly for request sizes after 128KB-256KB (reaching a rate of $r_{disk \rightarrow sram}$ =335MB/s, with 512KB request size for buffers in SRAM) while performance is suboptimal for smaller sizes: the degree of parallelism achieved with RAID0 is lower (fewer spindles fetch data into memory) and execution is dominated by overheads in the kernel's I/O subsystem. On the other hand, MBL delivers good performance even for small request sizes and comes close to the theoretical 528MB/s limit imposed by the PCI-X bus itself.

We note that throughput for transfers to RAM levels off early, at ∼217MB/s. We found this is due to an architectural constraint of the Intel motherboard used on Server B: The 6300ESB I/O hub supporting the PCI-X bus is connected to the 827210 Memory Controller through a "hub link" interface which is limited to 266MB/s. Server A's PCI bridge does not exhibit a similar issue (Fig. 3.7(a)).



(a) RAID and MBL, Server A

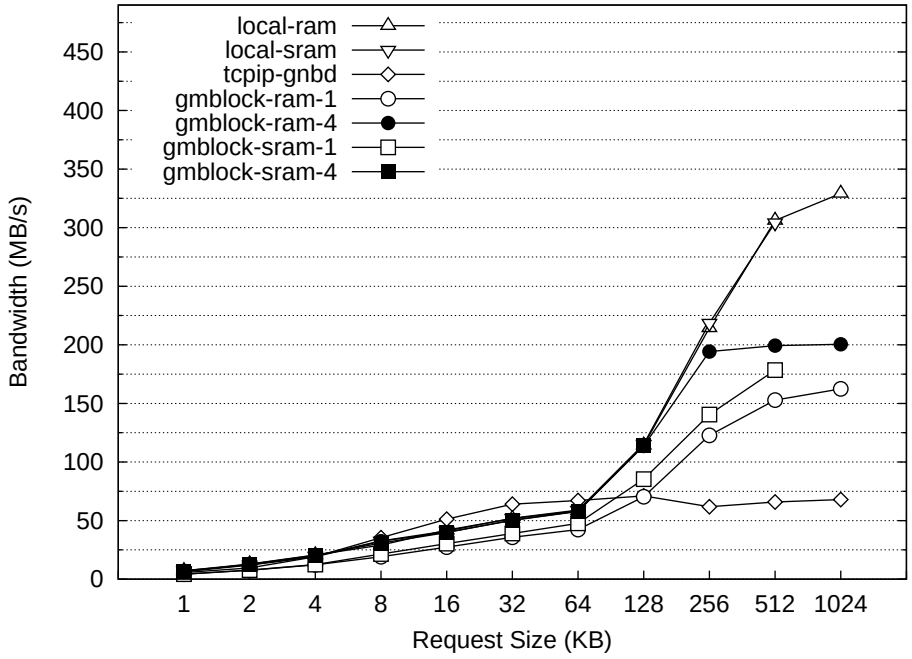


(b) RAID and MBL, Server B

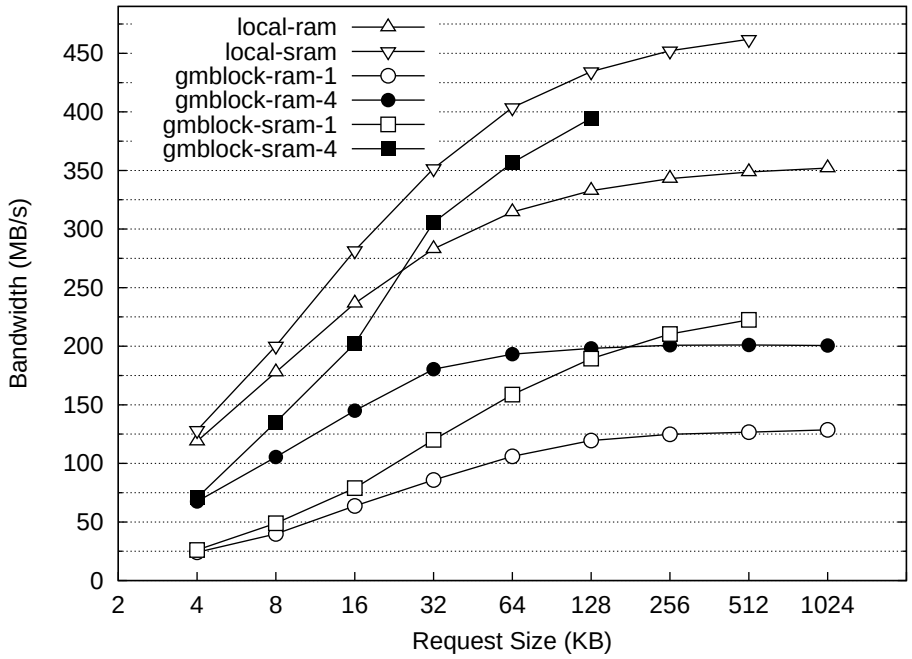
Figure 3.7: *Sustained local bandwidth for both storage media on Servers A and B*

3.3.2 Experiment 1b: Remote read performance

We then proceed to measure the sustained remote read bandwidth for all three implementations. A userspace client runs on a machine of configuration A generating back-



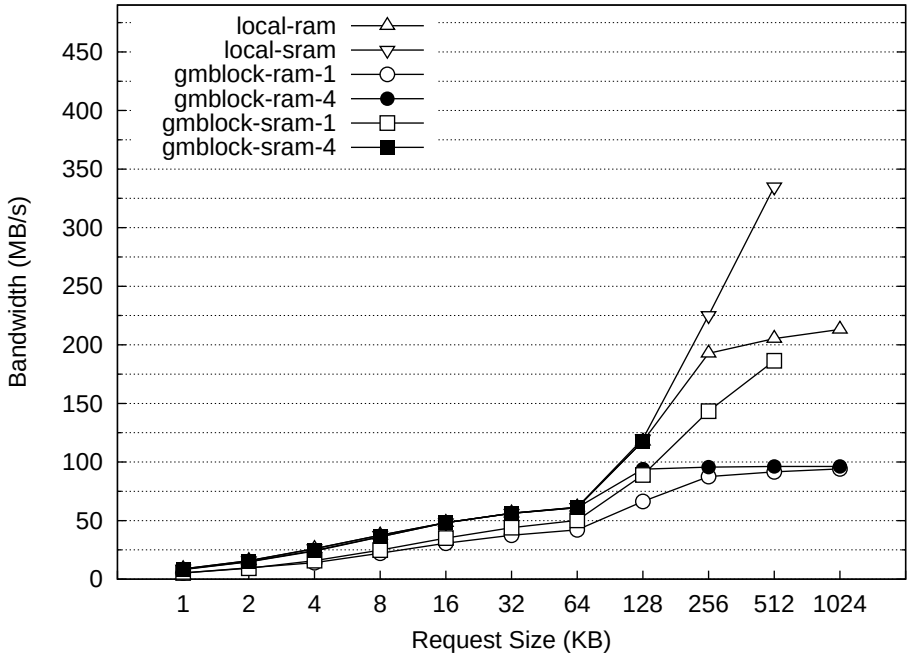
(a) RAID bandwidth



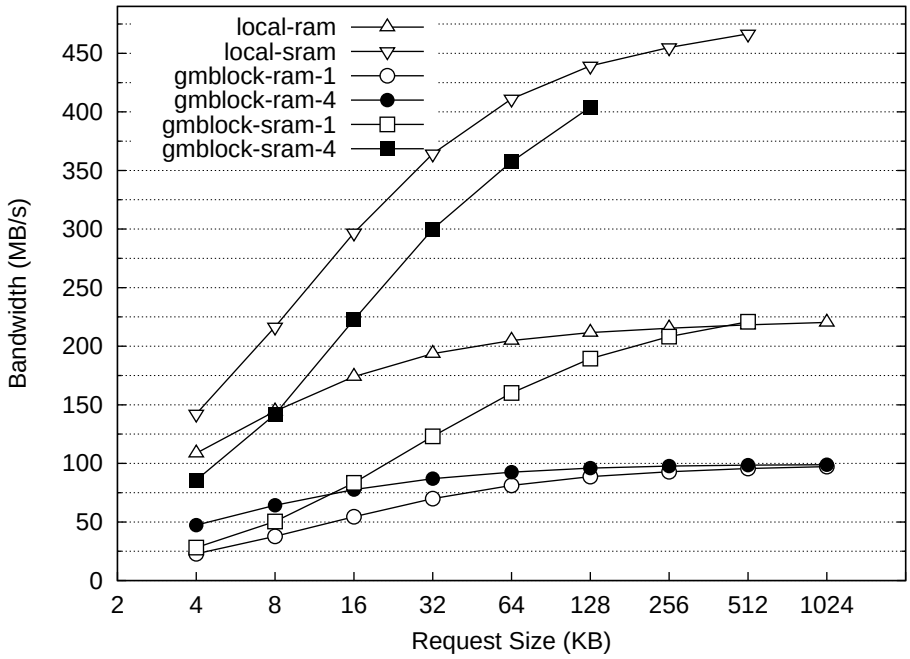
(b) MBL bandwidth

Figure 3.8: *Sustained remote read bandwidth, Server A*

to-back requests of variable size in two setups, one using Server A and one with Server B. To achieve good utilization of Myrinet's 2+2Gbit/s links it is important to pipeline requests correctly. We tested with one, two and four outstanding requests and the



(a) RAID bandwidth



(b) MBL bandwidth

Figure 3.9: *Sustained remote read bandwidth, Server B*

corresponding configurations are labeled `gmblock-ram-{1,2,4}` and `gmblock-sram-{1,2,4}` for the GM-based and short-circuit path case respectively. To keep the figures cleaner we omit the curves for `gmblock-{ram,sram}-2`. In general, the performance of

`gmblock-{ram,sram}-2` was between `gmblock{ram,sram}-1` and `gmblock{ram,sram}-4`, as expected.

We are interested in bottlenecks on the CPU, the memory bus, the RAID controller and the PCI/PCI-X bus. In general, GNBD performs poorly and cannot exceed 68MB/s on our platform; TCP/IP processing for GNBD consumes a large fraction of CPU power *and* a large percentage of memory bandwidth for intermediate data copying. A representative measurement is included in Fig. 3.8(a).

Performance for `gmblock-ram-1` and `gmblock-sram-1` is dominated by latency, since only one block read request is in flight at all times. Resource utilization is suboptimal, since the network interface is idle when the storage medium retrieves block data and vice-versa, hence the sustained throughput is low. The results are consistent with block data being transferred from the disk to buffers in RAM or SRAM (at a rate of $r_{disk \to ram}$ or $r_{disk \to sram}$ respectively), then from RAM or SRAM to the Myrinet fabric (at a rate of $r_{ram \to net}$, $r_{sram \to net}$ respectively). For example, in the case of `gmblock-sram` and the 3Ware controller on Server B (Fig 3.9(a)), $r_{disk \to sram}$ =335MB/s, $r_{sram \to net}$ =462MB/s and the expected remote read rate is

$$r_{gmblock-sram} = 1/\left(\frac{1}{r_{disk \to sram}} + \frac{1}{r_{sram \to net}}\right)$$

which is 194MB/s, very close to the observed value of 186MB/s.

When two or four requests are outstanding, the bottleneck shifts, with limited PCI-to-memory bandwidth determining the overall performance of `gmblock-ram`. In the case of Server B (Fig. 3.9(b)), `gmblock-ram` has to cross the hub link (266MB/s theoretical) to main memory twice, so it is capped to half the value of $r_{disk \to ram}$ = 217MB/s. Indeed, for request sizes over 128KB, it can no longer follow the local storage bandwidth curve and levels off at ∼100MB/s. This effect happens later for Server A's PCI host bridge, since its PCI to memory bandwidth is ∼398MB/s. Indeed, with request sizes over 128KB for MBL, `gmblock-ram` is capped at 198MB/s. On the other hand, `gmblock-sram` has no such limitation. In the case of MBL on Server A, `gmblock-sram-{2,4}` deliver more than 90% of the locally available bandwidth to the remote node for 256KB and 128KB-sized requests respectively, a two-fold improvement over `gmblock-ram`.

When used with a real-life RAID storage subsystem, `gmblock-sram-{2,4}` utilizes the

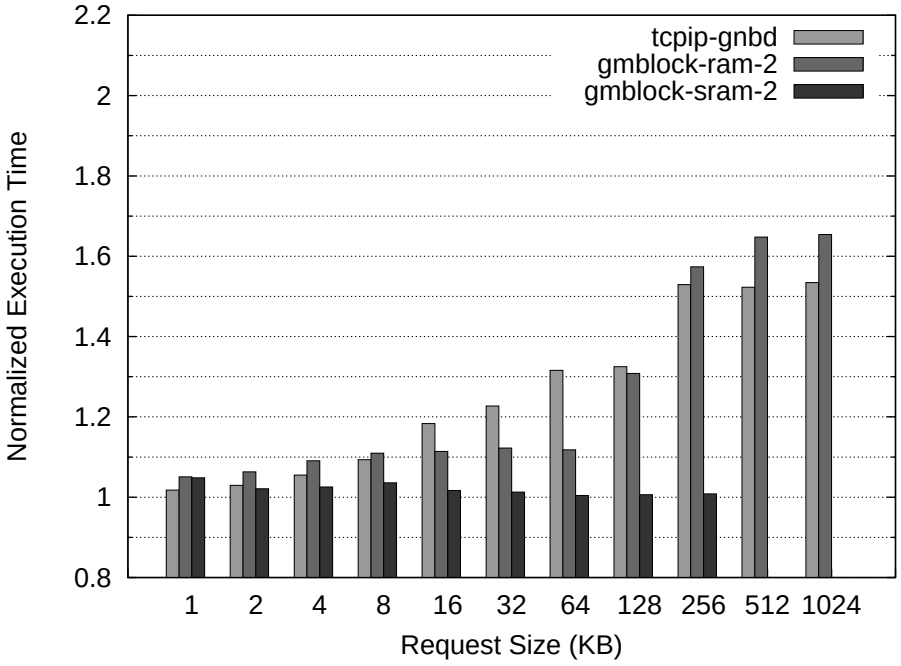


Figure 3.10: *Interference on the memory bus*

full RAID bandwidth for any given request size up to 256KB: for Server B that is 220MB/s, a more than two-fold improvement. For Server A that is 214MB/s, 20% better than `gmblock-ram`. Since `gmblock-ram` has already saturated the PCI-to-memory link, the improvement would be even more visible for `gmblock-sram-2` if larger request sizes could be used, since the RAID bandwidth cap would be higher for 512KB requests: a 512KB request equals the stripe size of the RAID array and is processed by all spindles in parallel. However, the maximum number of outstanding requests is limited by the amount of SRAM that's available for gmblock's use, which is no more than 700KB on our platform, thus it only suffices for a single 512KB request. The small amount of SRAM on the NIC limits the maximum RAID bandwidth made available to `gmblock-sram`. We propose synchronized GM operations to work around this limitation in Chapter 4.

In the case of the higher-performing MBL, we note that a higher number of outstanding requests is needed to deliver optimal bandwidth; moreover, a significant percentage of the maximum bandwidth is achieved even for small, 32-64KB request sizes.

It is also interesting to see the effect of remote I/O on the locally executing processes on the server, due to interference on the shared memory bus. Although `gmblock-ram` removes the CPU from the critical path, it still consumes two times the I/O bandwidth on the memory bus. If the storage node is also used as a compute node, memory con-

tention leads to significant execution slowdowns.

3.3.3 Experiment 1c: Effect on local computation

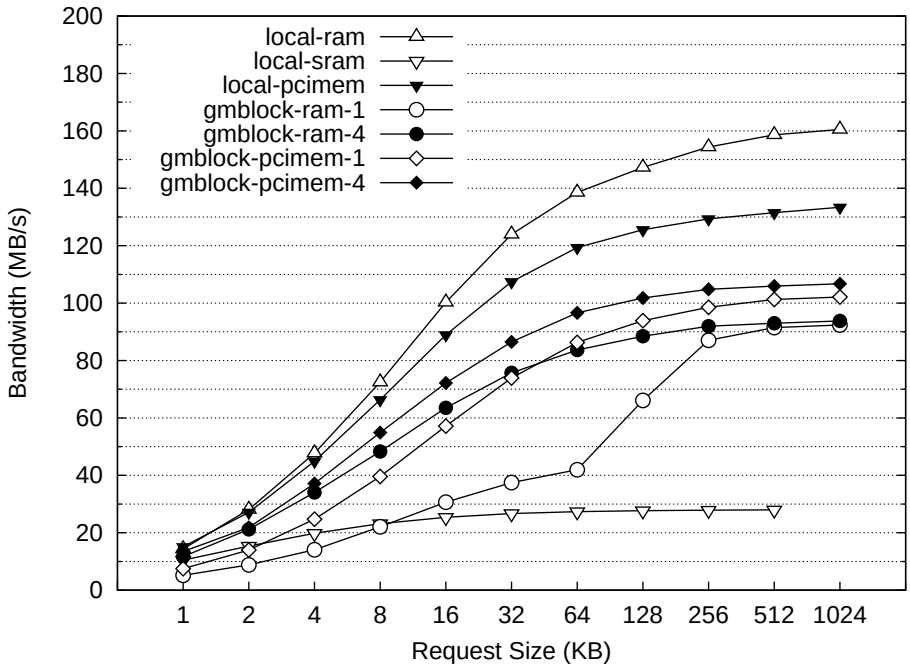
For Experiment 1c, we run `tcpip-gnbd`, `gmblock-ram-2` and `gmblock-sram-2` along with a compute intensive benchmark, on only one of the CPUs of Server A. The benchmark is a process of the `bzip2` compression utility, which performs indexed array accesses on a large working space (∼8MB, much larger than the L2 cache) and is thus sensitive to changes in the available memory bandwidth, as we have shown in previous work [KK06]. There is no processor sharing involved; the nbd server can always run on the second, otherwise idle, CPU of the system. In Fig. 3.10 we show the normalized execution time of `bzip2` for the three systems. In the worst case, `bzip2` slows down by as much as 67%, when `gmblock-ram-2` is used with 512KB requests. On the other hand, the benchmark runs with negligible interference when `gmblock-sram` is used, since the memory bus is bypassed completely and its execution time remains almost constant.

3.3.4 Experiment 1d: Remote write performance

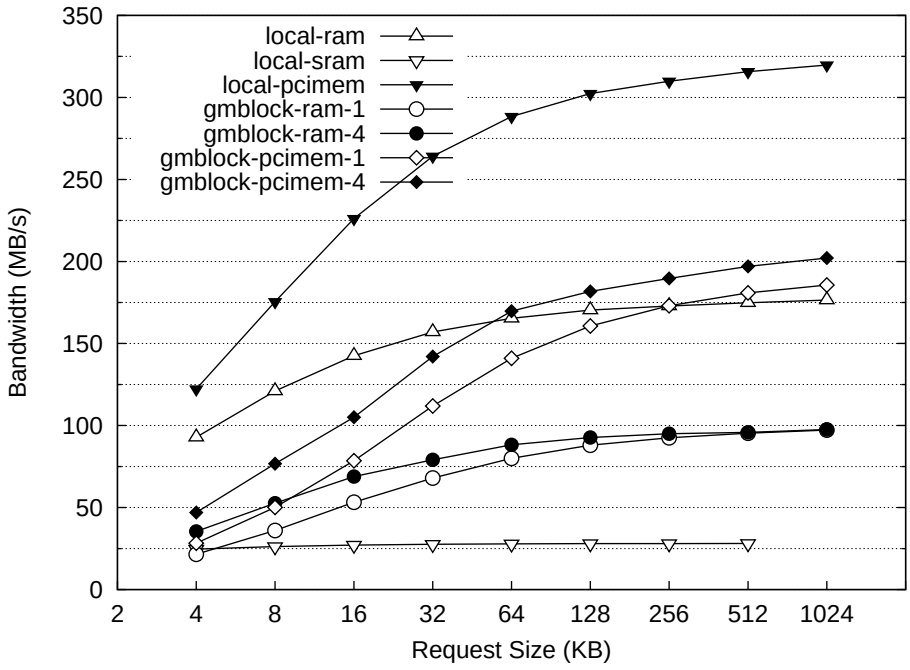
We also evaluate performance in terms of sustained remote write bandwidth, similar to the case of remote reads. The results for Server B are displayed in Fig. 3.11. Note that the maximum attained write performance of the RAID controller is much worse than the read case, however its bandwidth vs. request size curve rises sooner due to the use of on-board RAID write buffers.

Again `gmblock-ram` is capped at ∼100MB/s due to crossing the main memory bus. However, the performance of `gmblock-sram` is *much* lower than expected based on the read results. We discovered and later confirmed with Myricom this is due to a hardware limitation of the LanaiX processor, which cannot support being the target of PCI read transactions efficiently and delivers only ∼25MB/s PCI read bandwidth.

To work around this limitation, we introduce a third configuration; we can construct a data path which still bypasses main memory but uses intermediate buffers on the peripheral bus. As buffer space we decided to use memory on an Intel XScale-based PCI-X adapter, the Cyclone 740 [Cyc], placed in the 3rd slot of Server B's 3-slot PCI-X bus. It features an Intel XScale 80331 I/O processor and 1GB of DDR memory, on an internal



(a) RAID bandwidth



(b) MBL bandwidth

Figure 3.11: *Sustained remote write bandwidth, Server B*

PCI-X bus. A PCI-X to PCI-X bridge along with an Address Translation Unit allows exporting parts of this memory to the host PCI physical address space. By placing the message buffers of gmblock on the card, we can have the Lanai DMAing data *into* this

memory, then have the storage controller read data off it, which is more efficient than reading data off Lanai SRAM directly. This path crosses the PCI bus twice, hence is limited to half its maximum bandwidth. Moreover, even after careful tuning of the Intel XScale's PCI-X bridge to support efficient prefetching from the internal bus in order to serve incoming bus read requests, the 3Ware RAID controller was only able to fetch data at a rate of ∼133MB/s compared to ∼160MB/s from main memory (the `local-{ram, pcimem}` curves). Thus, the performance of the NIC ⟶ PCI buffers ⟶ storage path (`gmblock-pcimem`) is comparable to that of `gmblock-ram`, however it has the advantage of bypassing main memory, so it does not interfere with memory accesses by the host processor.

3.4 Discussion

3.4.1 Sharing of structured data

The proposed data path at the logical layer can be seen in Fig. 3.5(a). There are almost no gmblock-specific changes, compared to a GM-based nbd implementation. To achieve this, we re-use existing programming abstractions, as provided by GM and the Linux kernel. By building on `O_DIRECT`-based access to storage, our approach is essentially disk-type agnostic.

Since the CPU is involved in the setup phase of the transfer, the server is not limited to sharing raw block device blocks. Instead, it could share block data in *structured* form. The proposed framework concerns data *transport* without imposing limitations on how the storage server structures shared data, by decoupling the control path from the data path. The CPU may run filesystem code to locate the locations of affected blocks on disk(s), then invoke gmblock's mechanism to perform the transport on the short-circuit path. For example, storage for virtual machine images can manage them as files in a standard ext2 filesystem, and share them over gmblock.

Similarly, gmblock's approach can be integrated in an object-based storage scenario [Sun08, WUA+08] to enhance the operation of object storage servers: in the case of a Lustre OSS, object management code continues to run on the host CPU in order to traverse filesystem structures, with minimal interference by data transport operations.

Protocol operations are undertaken by the host CPU, while file data are moved over the proposed path, directly between network and storage devices.

3.4.2 Caching and prefetching

Another point to take into account is ensuring coherence with the kernel's page cache. Since blocks move directly from NIC to storage and vice versa, a way is needed to ensure that local processes do not perform I/O on stale data that are in the kernel's page cache and have not been invalidated. We avoid this problem by keeping the kernel in the processing loop, but not in the data path. Its direct I/O implementation will take care of invalidating affected blocks in the page cache, if an `O_DIRECT` write takes place, and will flush dirty buffers to disk before an `O_DIRECT` read.

Using the proposed path means that server CPU and RAM are no longer in the processing path. Although this eliminates architectural bottlenecks, it means no server-side prefetching and caching is possible. Moreover, overall performance depends on the interaction of gmblock with the overlying filesystem and application, and the amount of read-write sharing that takes place through shared storage. We discuss the importance of these three factors, prefetching, caching and read-write sharing below.

Server-side buffers on storage servers may play an important role in prefetching data from the storage medium for efficiency. Data prefetching is still possible in gmblock, but has to be initiated by the client side. Experimental evaluation on an OCFS2 over gmblock setup (Section 5.4) shows that prefetching is indeed necessary, to support good performance for small application reads.

Regarding caching, although server-side caching is no longer possible, client-side caching still takes place: Client systems treat gmblock-provided storage as a local hard disk, caching reads and keeping dirty buffers on writes. Data need to be written back only to ensure correctness for read-after-write sharing. Moreover, the aggregate size of memory in clients can be expected to be much larger than storage server memory.

Server-side caching may also prove to be beneficial in coalescing small reads and writes, forming more efficient requests to storage. The degree of coalescing is highly dependent on the overlying application's access patterns. Whether using the proposed data path leads to performance increase depends on the balance of memory-to-memory

copy throughput, imposed CPU load and memory-to-storage transfers, relative to direct storage-to-network throughput.

Finally, server-side caching may prove important for applications with very heavy read-write sharing through the filesystem. If there is no special provision for direct client-to-client synchronization of dirty data, either by the application itself or by the filesystem, then the storage server may be used essentially as a buffer for client-to-client block transfers with many small writes followed by many small related reads. For such workloads, using gmblock's short-circuit data path is not an appropriate choice. If write coalescing is needed, then the framework can be set to cache writes, while still allowing reads to happen over the direct data path, or it can be run in fully cached mode. The same block device can be exported through a number of servers in different modes, simultaneously; the short-circuit path is coherent with page cache accesses, since the kernel remains in the setup phase of the transfer.

On the other hand, the impact of heavy read-write sharing in application performance is a well-known problem and there are numerous efforts in the literature to attack it at the filesystem and application level. Oracle 9i uses a distributed caching mechanism called "Cache Fusion" [LSC+01] to support read and write-sharing with direct instance-to-instance transfers of dirty data over the interconnect, instead of going through shared storage. Disk I/O happens only when the needed blocks are not present in *any* of the client-side caches. Similarly, in the HPC context, heavy read-write block contention arises in MPI applications with multiple peers working on the same block set. To attack the problem, GPFS features a special data shipping mode [PTH+01], which the free ROMIO implementation of MPI-IO has also been extended to support [BICrT08]. Data shipping mode minimizes read-write block contention by assigning distinct parts of a shared file to distinct processes, so that only a single process issues read/write requests for a specific block during collective I/O.

To summarize, the applicability of gmblock's direct I/O path depends significantly on the access patterns of the I/O workload. Workloads for which it is a good fit are those with little data sharing, e.g., shared storage for live VM migration and read-write server workloads on independent datasets, or workloads where nodes can be assumed to coordinate access at a higher level, e.g., the Oracle RDMBS or MPI-IO applications with data shipping optimizations.

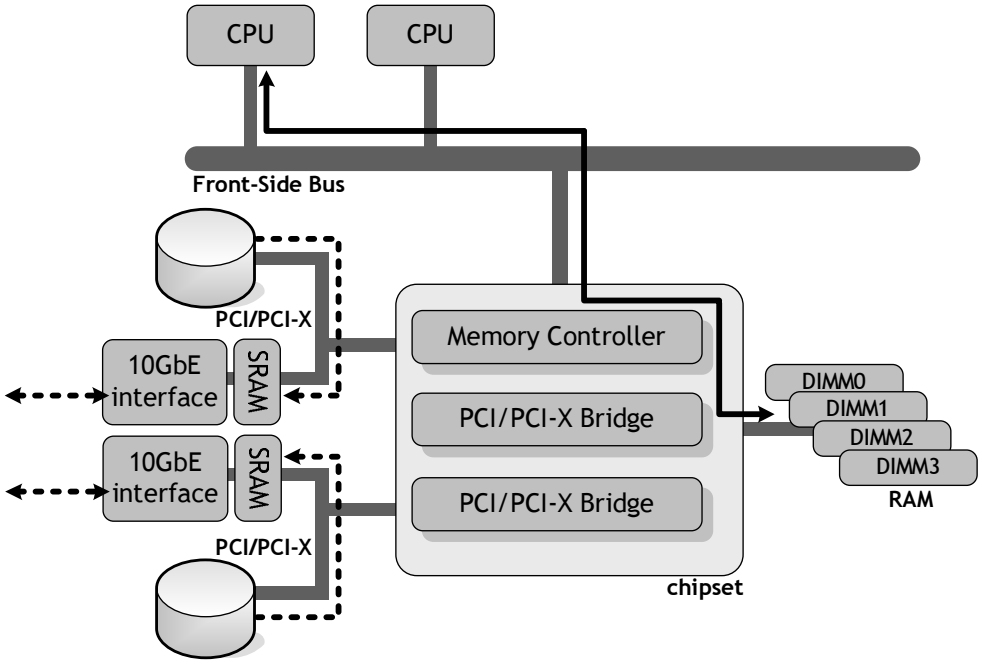


Figure 3.12: *SRAM placed between the peripheral bus and a commodity interface*

3.4.3 Applicability to other interconnection technologies

Our prototype implementation is based on Myrinet, which offers a fully programmable NIC. Availability of such NIC provides flexibility, and allows for almost all of the communications protocol processing to take place on the NIC, minimizing host CPU overhead. However, it is not necessary for constructing the proposed data path.

The prerequisites for server-side gmblock operation are DMA-enabled storage, a DMA-enabled network interface and a small, fast, PCI-addressable memory area *close* to it. Thus, to make our approach applicable to interconnection technologies such as Infiniband and 10-Gigabit Ethernet, we need to amend the hardware to include a small amount of memory between the peripheral bus and the network interface. Fig. 3.12 displays such setup; a small amount of SRAM placed between the peripheral bus and a commodity 10GbE interface enables two disk-to-NIC paths to work in parallel.

To transfer data for a remote block read request, the host CPU initiates a DMA operation from the storage device to the PCI physical addresses corresponding to this SRAM. The degree of CPU involvement during the network send operation depends on the capabilities of the NI used. In the case of TCP/IP and commodity Ethernet, the protocol stack will initiate a gather DMA operation to collect frame headers from host RAM and frame data from the SRAM and inject them into the network. All of the protocol pro-

cessing happens on the CPU, which will have to manage individual frame completions, via a combination of interrupts and polling.

In the case of Infiniband, the corresponding SRAM regions can be registered with the NI and the data sent to the requesting node over RDMA. The RDMA operation progresses independently of the host CPU, which is notified when all of the message data have been transferred to the remote node.

3.4.4 Applicability to low-power designs

The proposed approach is also relevant with the emerging trend for ubiquitous, low-power, embedded storage devices based on a System-on-Chip (SoC) architecture. These devices typically feature lower clock rates, typically in the 500MHz-1GHz range, and lower-bandwidth internal buses, for reasons of cost and power efficiency. Staging data for I/O operations in memory buffers exacerbates memory bandwidth pressure; peer-to-peer data transfers between the NI and storage devices make the path to main memory available for use by protocol processing on the CPU and enable the storage device to scale better with disk count and network/storage bandwidth. Furthermore, staging data in an appropriately-sized SRAM instead of in the main memory of the host, which is typically implemented with SDRAM or DDR, can effect significant performance savings [MACM05, ACMM07]. Lowering DDR bandwidth requirements also allows for aggressive power optimizations; a low-stress DDR266 workload consumes less than 17% of the power needed for a high-stress DDR333 workload [Jan01].

Certain SoCs already feature a small amount of SRAM; for example, the Intel IOP80332 [Int] features an on-chip SRAM controller which can manage up to 1MB of SRAM.

4 Synchronized GM send operations

4.1 Motivation

Overall, gmblock delivers significant bandwidth improvements for remote read / write operations compared to conventional data paths crossing main memory. However, its performance still lags behind the limits imposed by network and local storage bandwidth; in the case of MBL on Server B, `gmblock-sram` achieves ∼400MB/s (for 128KB-sized requests in the `gmblock-sram-4` case, which is 86% of the maximum available read bandwidth available locally, while it achieves ∼220MB/s (65% of the maximum) when using a real-life RAID-based storage system. The chief reason for low efficiency lies in the interplay between a number of conflicting factors and architectural constraints: *(a)* The nbd system needs an outstanding request queue of sufficient depth in order to pipeline requests efficiently and keep all system components busy, *(b)* RAID-based storage systems exhibit a bandwidth vs. request size curve as shown in Fig. 3.7(a); they deliver their best performance when provided with sufficiently large requests which can be parallelized across all spindles, *(c)* Cluster interconnect NICs, such as the Myrinet NICs on our platform, feature a limited amount of memory, which is usually only meant to buffer message packets before injection into the network.

The proposed approach moves data directly from disk to NIC, thus the maximum number of outstanding requests is limited by the amount of SRAM available for gmblock's use. In our case, using standard Myrinet NICs with 2MBs of SRAM, we were able to reserve ∼700KB for gmblock buffers. Allocating more than that is not a viable option in production environments, because there is a number of side effects:

- a number of GM features, such as Ethernet emulation, have to be disabled in order to reduce the size of the firmware.

- less SRAM is available for caching virtual-to-physical translations on the NIC. Reducing the size of the translation cache increases the translation miss rate and may impact messaging performance significantly, depending on the degree of buffer reuse.

- the maximum supported number of nodes is reduced, since less space is available for GM to hold connection state between pairs of nodes.

This means that only 1×512KB request or 2×256KB or 4×128KB requests may be in flight at any moment.

Using a larger number of smaller requests improves pipelining but moves us at a lower point on the request size vs. bandwidth curve of fig. 3.7(b). This becomes more evident as the number of disks in the RAID array increases. When the number of outstanding requests is low, the system is dominated by latency.

To allow gmblock to use larger requests, while still achieving good disk and network utilization, we focus on increasing the amount of overlapped processing *within* each individual request. We aim to have the network send data for a remote block read request even before the storage medium has finished serving it. This way, we can take advantage of the full bandwidth of the RAID controller while still having good overlapping of disk with network I/O, even with a single, large request on the fly.

4.2 Design

Servicing a block read request entails two steps: Retrieving data from disk to SRAM, then sending data from SRAM to the network. To enable intra-request overlapping, the send from SRAM operation needs to be *synchronized* with the disk read operation, in order to ensure that only valid data are sent over the network. Ideally, this should be done with minimum overhead, in a portable, block device driver-independent way and with minimal changes to the semantics of the calls used by the nbd server for local and network I/O.

There are several approaches to implementing this kind of synchronization mechanism in software, inside the nbd server, in order to submit large I/O requests to local storage while splitting them up for processing by the network.

One approach is to uncouple the size of block I/O requests from the size of network I/O requests. The server may divide each large remote read request of l bytes (e.g., 1MB) in much smaller chunks of c bytes (e.g., 4KB), then submit them all to the underlying storage medium for processing. A userspace nbd server can use the POSIX Asynchronous I/O facility to have multiple outstanding requests to local I/O.

This scenario has a number of significant drawbacks. First, it incurs significant request processing overhead, since the server receives individual completion notifications per block I/O and network I/O request, and invokes individual GM send calls. Similarly, on the client side, the host CPU is frequently woken up on the critical path to acknowledge reception of individual GM messages and facilitate the flow of GM send and receive tokens. Second, this approach discards the information that all chunks are contiguous; instead it relies the server-side I/O scheduler to reassemble them into a larger I/O request to the storage medium, for efficiency.

We propose a synchronization mechanism working directly between the storage medium and the Myrinet NIC, in a way that does not involve the host CPU and OS running on top of it at all, while at the same time remaining independent of the specific type of block storage device used.

Let us consider the scenario when the server starts a user level send operation before the actual `read()` system call to the Linux I/O layer. This way, sending data over the wire is bound to overlap with fetching data from block storage into the Lanai SRAM. However, this approach will most likely fail, since the correctness of the data being transmitted depends on the ability of the storage medium to deliver data faster than the Send DMA engine on the NIC consumes them. The disks may fail to keep up for a variety of reasons, such as transient disk errors, concurrent access by applications other than the nbd server, or simply because it cannot deliver enough bandwidth to saturate the network link(s).

To solve this problem we introduce the concept of a *synchronized* property for user level send operations. A synchronized GM operation ensures that the data to be sent from a message buffer are valid, before being put on the wire. If at some point in time no valid

data are available for a synchronized send token, the firmware ignores it when searching for a suitable token from which to enqueue a packet to the network. Essentially, the NIC works in lockstep with an external agent (the block device), throttling its send rate in order to match that of the incoming data (in our case, r_{disk}).

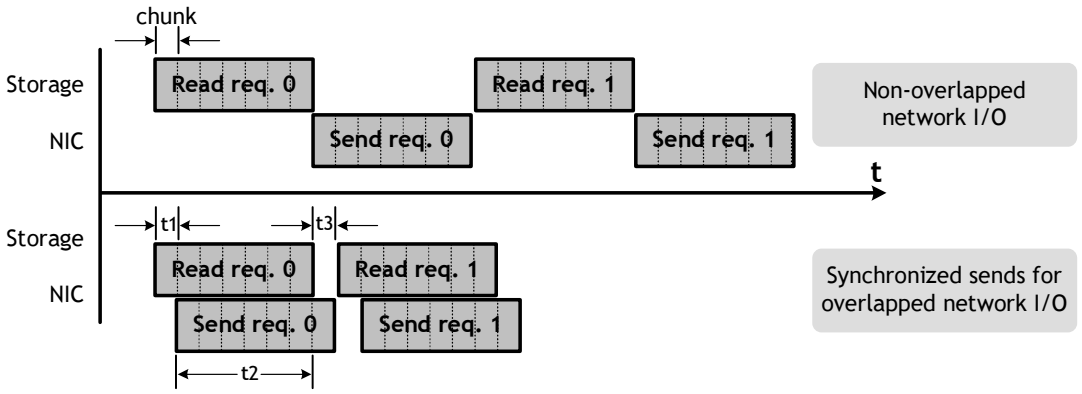


Figure 4.1: *Intra-request phase overlap*

The NIC notices data transfer completions in chunks of c bytes. The value of c determines the synchronization grain and the degree of overlapping achieved (see Fig. 4.1); The NIC only starts sending after $t_1 = \frac{c}{r_{disk}}$ time units, then both the storage device and the NIC are busy for $t_2 = \frac{l-c}{r_{disk}}$, then the pipeline is emptied in $t_3 = \frac{c}{r_{net}}$ time units. There is a trade-off involved, since smaller values mean finer-grained synchronization and better overlapping, but could impose significant CPU overhead on the Lanai, while bigger values lead to lower synchronization overhead but reduce the overlapping between the two phases.

The semantics described above break the assumption that the whole of the message is available when the send request is issued, allowing the NIC to synchronize with an external agent while the data are being generated.

However, as we discovered experimentally and explain in greater detail in section 4.4, this does not suffice to extract good performance from our platform; the design retains the assumption that the external agent places data into the message buffer as a single stream, in a sequential fashion. However, most real-world storage devices rely on *parallelizing* request processing to deliver aggregate high performance, e.g., by employing RAID techniques. Thus, incoming data comprise *multiple* slower streams – Fig. 4.2 shows a 4-disk RAID0 array DMAing a single RAID stripe.

To achieve high degree of overlapping, we need to incorporate this multiplicity of streams

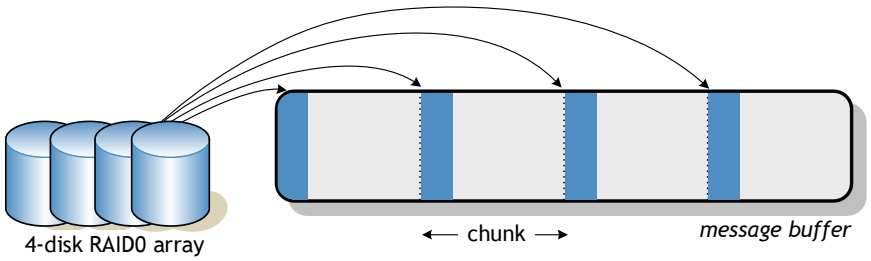


Figure 4.2: *A 4-disk RAID0 array moving a stripe into a message buffer*

in the semantics of synchronized operations and the design of the networking protocol. We extend reliable messaging to allow the sender to construct network packets from disjoint locations inside the message buffer, and the receiver to support out-of-order placement of incoming fragments.

The cost of having the sender-side NIC notice DMA transfer completions anywhere inside the message is prohibitive, hence we make a compromise: we support "multiple stream" synchronized operations by having the user provide *hints* on the position and length of a finite number of incoming streams inside the buffer. In our case, they are derived from RAID array member count and chunk size.

Although our prototype implementation of synchronized operations is Myrinet/GM based, it is portable to any programmable NIC which exposes part of its memory onto the PCI address space and features an onboard CPU. Synchronization happens in a completely peer-to-peer way, over the PCI bus, without any host CPU involvement.

4.3 Implementation issues on Myrinet/GM

There is one major implementation-specific point which has not yet been addressed. We need a way for the Lanai to be notified as an external agent places data into its SRAM. However, the Myrinet NIC does not provide such functionality in hardware. It could be implemented with a "dirty memory" bitmap describing the state of the Lanai SRAM (2MBs) divided into chunks of size c bytes. The bit corresponding to an SRAM chunk is set by the hardware whenever a value is written anywhere into it. For a value of $c = 4096$ bytes, at most 64 bytes are needed. The firmware initializes all bits corresponding to an SRAM buffer involved in a synchronized operation to zero. Then, it can verify chunk n contains valid packet data by checking if the bit for chunk $n+1$ is set, assuming that the

external agent performs DMA into the SRAM serially in ascending order of addresses for a single message chunk.

Since such functionality was not available on our NICs, we emulated it in software, with 32-bit markers in the SRAM itself. The Lanai polls the markers, which get overwritten as the data are DMAed in. The probability of at least one overwritten marker going undetected because the value being sent coincides with the magic value being used is very low. For instance, for a random block of l =1MB and a value of c =4KB, it holds:

$$P_{no\_ovr} = 1 - \left(1 - 2^{-32}\right)^{\lceil \frac{l}{c} \rceil} \Longrightarrow P = 5.96 \times 10^{-8}$$

.

Still, to ensure that the system never fails and the Lanai does not loop infinitely around a marker, an extra one is used right after the end of the block. It is set by the host CPU, i.e., by the nbd server application when the data transfer into SRAM is complete and all of the data is valid. The worst-case scenario is that overlapping does not take place, with probability P_{no_ovr} and the network transfer starts after the block read operation is complete.

The implementation of synchronized operations on Myrinet/GM comprises three phases:

Initialization phase Function `gm_synchro_prepare_buffer()` writes a 32-bit value, `GM_SYNCHRO_MAGIC`, aligned with the end of each chunk, once every `GM_MTU` bytes in the SRAM buffer. Initialization is done with PIO inside the GM library, since the host CPU is an order of magnitude faster than the Lanai. Send events passed to the firmware are flagged as synchronized by `gm_synchro_send_with_callback()` and may optionally define the offsets and lengths of multiple streams inside the message buffer. Similarly, send tokens created by the SDMA machine are enhanced to include send flags and multiple stream hints.

Transmission phase This step lies in the critical path of transmission and is executed whenever a new packet of at most `GM_MTU=4096` bytes is about to be injected into the network. Upon encountering a synchronized send token, the SDMA machine scans individual streams until one with valid data is found. If no stream contains a packet with valid data yet, the send token is bypassed. The firmware never blocks on a synchronized operation, ensuring fairness between tokens.

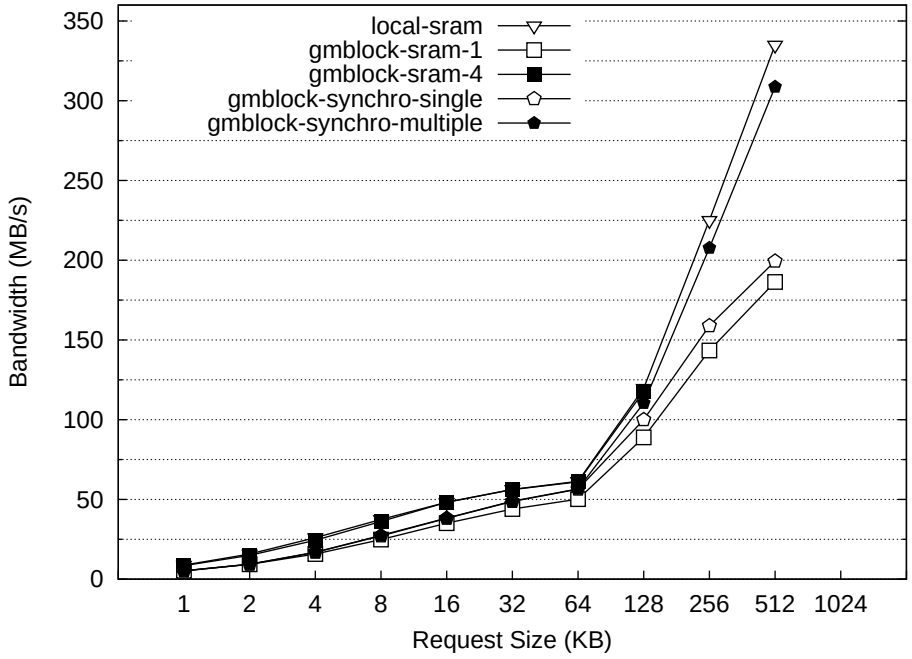
Finalization phase When the block I/O request completes, the nbd server notifies the GM firmware that all of the data are valid by calling `gm_synchro_finalize_buffer()`. This sets the marker right after the end of the block buffer to a magic value.

For multiple stream support, we modified GM's Go Back N protocol so that out-of-order construction of message packets at the sender side and placement at the receiver side is possible. Whenever a packet is injected into the network, the SDMA state machine keeps track of stream-specific state inside the send record being created; this information helps associate the send record with the stream from which the data for the packet originated. Should the connection be rewound due to lost packets or timeouts, the sender's RECV machine will know which of the stream-specific send pointers to modify in order to resend. Similarly, the RDMA state machine does not assume incoming message fragments are to be placed serially inside the matched buffer; we extended the GM packet header so that an optional offset field, `h_synchro_ptr` is used to determine where to DMA incoming fragment data inside the message buffer.

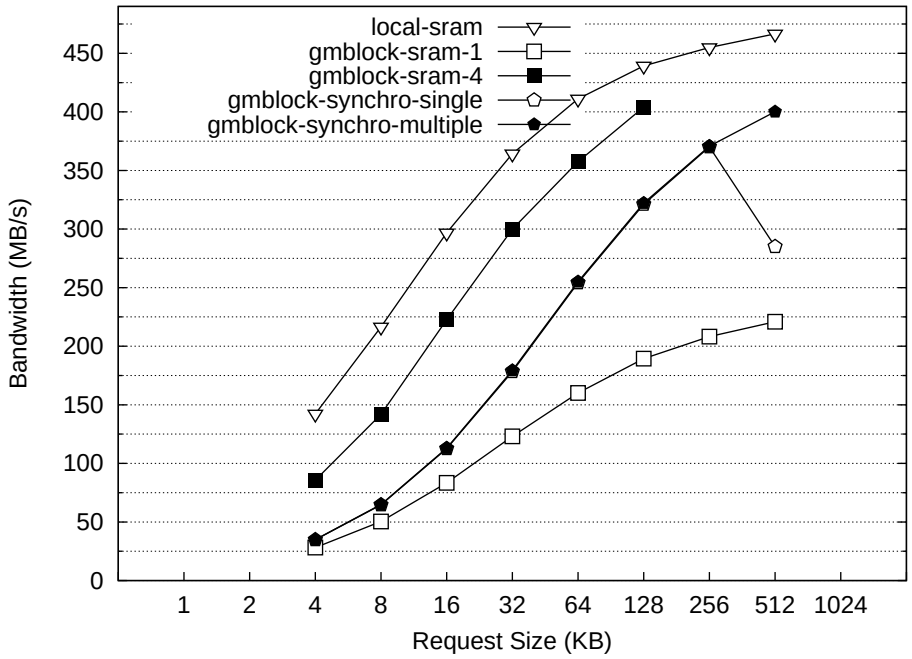
The Lanai cannot address host memory directly but only through DMA. The cost of programming the PCIDMA engine in order to monitor the progress of a block transfer to in-RAM buffers is prohibitive, so synchronized GM operations are only available when sending from buffers in Lanai SRAM. However, this suffices for implementing an optimized version of gmblock's data path.

4.4 Experimental evaluation

This section presents an experimental evaluation of gmblock extended to support synchronized operations versus the base version of gmblock using a direct disk-to-NIC data path. We do not include any instances of `gmblock-ram-{1,2,4}` since we have already shown how staging data in RAM-based buffers is detrimental to overall performance. The results were taken on Server B because it features a better performing, PCI-X bus.



(a) RAID controller, sustained read bandwidth



(b) MBL, sustained read bandwidth

Figure 4.3: *Sustained remote read bandwidth for single and multiple-stream synchronized operations*

4.4.1 Experiment 2a: Synchronized Send Operations

We use two versions of gmblock: `gmblock-synchro-single` issues single-stream synchronized operations with one outstanding request, while `gmblock-synchro-multiple` supports multiple-stream synchronized operations. The number of streams is set equal to the number of disks in the RAID array. The results for both storage media are displayed in Fig 4.3(a), Fig 4.3(a) for the 3Ware Controller and the MBL device respectively.

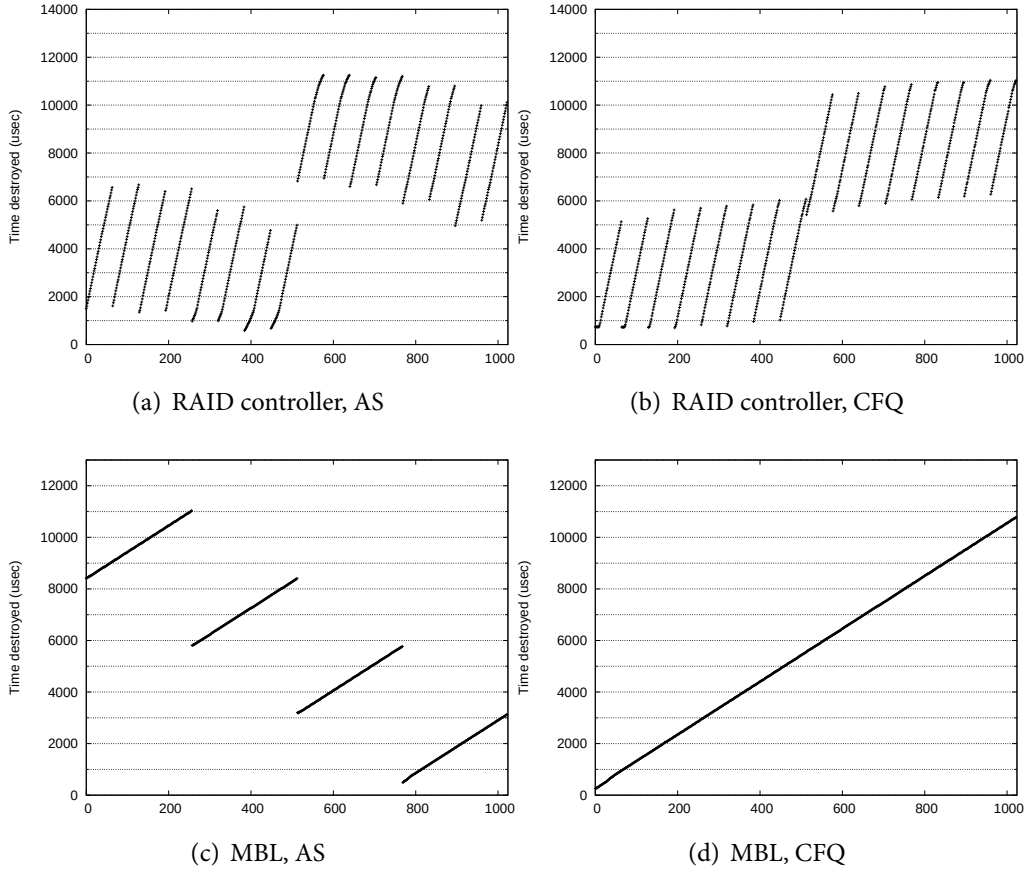
As expected using `gmblock-synchro-single` yields much better throughput over `gmblock-sram` for MBL (370MB/s, a 77% improvement, for 256KB-sized requests). Even with a single outstanding request `gmblock-synchro-single` reaches 91% of the maximum read throughput of `gmblock-sram-4`, by improving the latency of individual requests (e.g., 59% for 64KB-sized requests). This is a sharp drop-off in performance for 512KB-sized requests, which we focus on shortly. Contrary to MBL the performance gains of `gmblock-synchro-single` for the RAID configuration are marginal (7% improvement over `gmblock-sram-1`, for 512KB-sized requests).

4.4.2 Experiment 2b: RAID data movement

To better understand the reasons for the performance drop-off for 512KB MBL requests and the rather low performance of the RAID configuration, we need more insight on the way DMA operations progress over time. We use a custom utility, `dma_poll`, which provides data movement traces using predefined marker values (fig 4.4), similarly to the method described in Section 4.2. This way we can monitor when each individual chunk is DMAed into the message buffer. We find two reasons behind the results of the previous section:

RAID data movement `gmblock-synchro-single` ignores the fact that data are placed in different parts of the message buffer in parallel (see fig. 4.4(b)) and only overlaps disk and network I/O for the first chunk.

Segment reordering The maximum hardware segment size for DMA operations with the MBL device is 256KB. For requests greater than that, the actual order that the segments are submitted depends on the Linux I/O scheduler. Using the anticipa-



(a) RAID controller, AS (b) RAID controller, CFQ

(c) MBL, AS (d) MBL, CFQ

Figure 4.4: *DMA traces for the anticipatory and CFQ schedulers, 1024KB-sized requests*

tory I/O scheduler leads to the two segments being reordered for 512KB requests (fig. 4.4(c)). Hence the degree of overlapping for `gmblock-synchro-single` is lower.

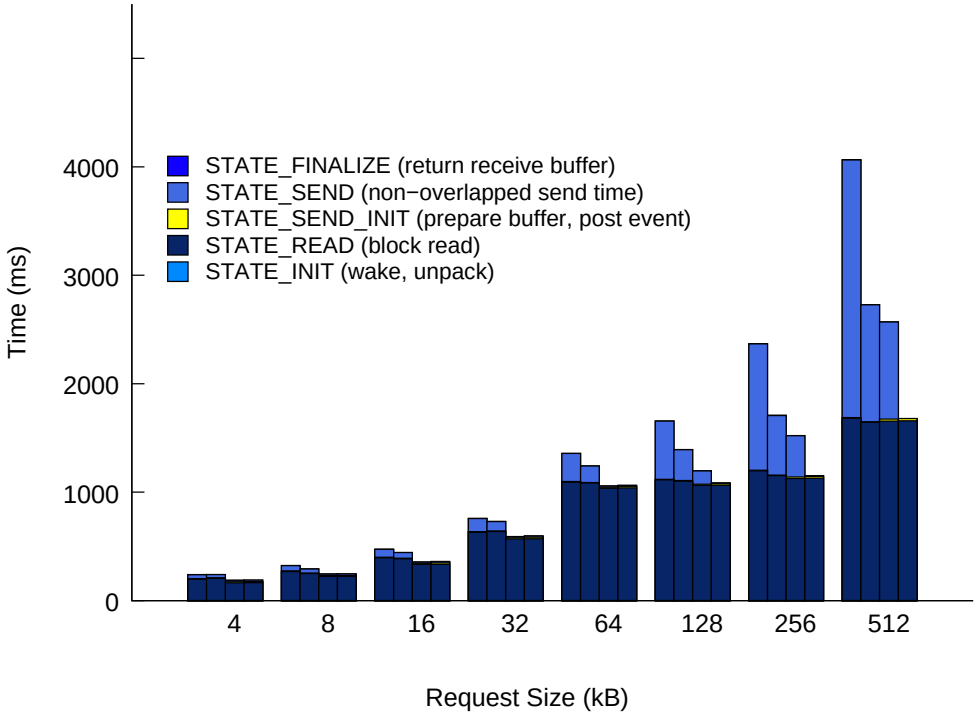
On the other hand, using `gmblock-synchro-multiple` works around these problems and achieves read rates close to that of local access. In the case of the 3Ware controller it reaches 92% of the maximum bandwidth (40% better than `gmblock-sram-4`) achieving near-perfect overlapping.

To demonstrate how overlapping influences request processing, we break down the total request processing time into distinct phases. The server monitors the state of each request as it makes progress, using counters based on the TSC register of the Pentium. Fig. 4.5 shows the time spent by the gmblock server in various stages of request processing. We identify five different states:

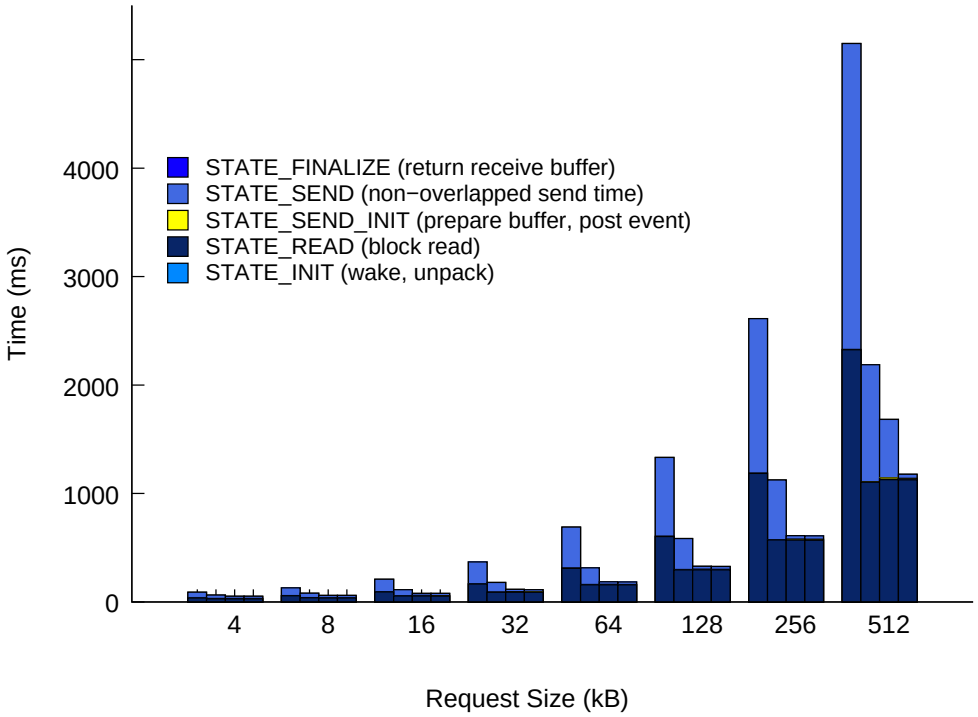
- `STATE_INIT`: A new nbd request has been received and is being unpacked.

- `STATE_READ`: A block I/O request is issued and the nbd server waits for it to complete.
- `STATE_SEND_INIT`: The nbd server posts a new GM send event to the Lanai.
- `STATE_SEND`: Disk I/O has completed and the GM send operation is in progress.
- `STATE_FIN`: Completion of send operation, returning request receive token back to GM.

The time spent on states other than `STATE_READ` or `STATE_SEND` was found to be negligible. `STATE_SEND` represents network I/O time that was not overlapped with disk I/O, and indicates the degree of disk to network I/O overlapping achieved. The integration of synchronized operations in `gmblock-synchro-multiple` makes it negligible.



(a) RAID controller



(b) MBL

Figure 4.5: *Latency breakdown per request size for gmblock-{ram, sram, synchro-single, synchro-multiple}*

5 Client-side issues and end-to-end evaluation

In the previous chapters, the primary focus has been on server-side optimizations to the gmblock data path. This chapter presents the client-side design and implementation of gmblock. We begin with a basic kernelspace client, which presents a virtual block device to the host system, then examine the various implementation choices, and their efficiency with regard to data movement.

We exploit Myrinet NIC programmability to enhance the operation of the kernelspace client. We propose extensions to GM networking to support scatter-gather I/O directly to/from the physical address space, and integrate them in the gmblock client. The end-to-end performance of the integrated system is evaluated with application benchmarks running on a parallel filesystem deployed on top of gmblock.

5.1 Design considerations for a kernelspace nbd client

5.1.1 Principles of operation

Let us return to Fig. 3.1(a), which presents the basic design of an nbd system. On the client side, the nbd client runs in kernelspace, and implements the host OS's block device driver interface to export a virtual block device. Device access requests originate either in applications performing raw block I/O, or in the filesystem layer. The requests are processed by the client-side block device layer, which will perform request coalescing

and I/O scheduling before passing them to the nbd client. This process is described in greater detail in Section 2.3.

The nbd client has to encapsulate the request in one or more network messages which are sent to the remote nbd server. When a reply arrives, the client notifies the overlying block device layer for I/O *completion*. The block device layer will then wake up any code waiting for the result of the I/O request, so that it can resume queueing I/O and the cycle can begin again.

5.1.2 Client-side data movement

The nbd client needs to gather data from block I/O buffers into network packets, in the case of write requests, and scatter incoming data to block I/O buffers, in the case of read requests. To improve nbd performance on the client side, we need to minimize the amount of redundant block data movement while translating between host block I/O requests and network messages.

I/O buffers reside in host RAM and are managed either by the application or by the kernel itself. There are two distinct cases:

- An application uses the raw block device directly and employs custom caching and prefetching policies using direct I/O from/to virtually addressed userspace buffers. I/O operations refer to areas contiguous in the application's VM space, which are mapped to disjoint areas in physical memory. The direct I/O layer of the host system takes care of enforcing memory access restrictions, performing virtual-to-physical translation and *pinning* the relevant pages in physical memory, before passing the request to the nbd client.
- An application uses the raw block device for buffered I/O, or a filesystem is mounted on top of the virtual block device. In this case, the nbd client is called to perform I/O to/from pages in the page cache, as the kernel deems necessary.

In either case, the nbd client is presented with a scatter-gather list of pages in physical memory, to be filled with data, or to be flushed to disk. The amount of data movement necessary after this point, and the corresponding I/O overhead depends significantly on the capabilities of the underlying network.

The situation is similar to that described in Section 3.1.1 for the nbd server. If a TCP/IP-based interconnect is used, then the nbd client will have to copy data in and out of socket buffers and invoke the TCP/IP stack. The imposed overhead depends on the quality of the TCP/IP implementation and whether the NIC offers TCP offload capabilities.

Let us study the interaction of the nbd client with a user level networking infrastructure. We will focus on Myrinet, since this is the substrate used for the prototype gmblock implementation, but the conclusions drawn hold for user level networking interfaces in general.

Even though the Myrinet NIC undertakes most of network protocol processing in Myrinet, translation between large block I/O requests referring to scattered physical memory pages and GM network messages is not straightforward; there is discrepancy between the needs of a kernelspace-based nbd client implementation and the capabilities of GM.

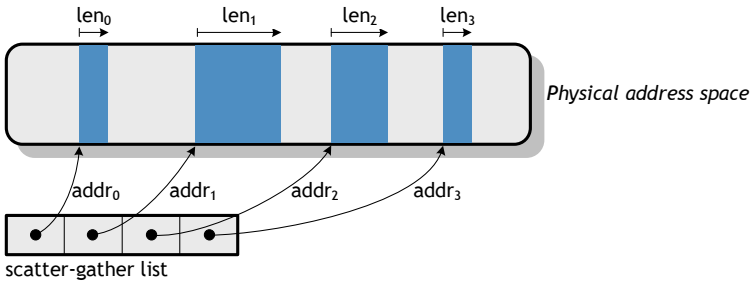


Figure 5.1: *A scatter-gather list describes an I/O buffer dispersed in physical memory*

GM requires all communication to happen from/to pre-registered message buffers which are contiguous in the application's VM space. Scatter-gather I/O happens via address translation; GM keeps track of message progress using virtual pointers and translates from the contiguous VM area to possibly disjoint physical pages. On the other hand, the kernelspace nbd client is presented with discontiguous segments in *physical* memory; block I/O buffers are described in terms of scatter-gather lists, referring to a number of disjoint physical memory segments (Fig. 5.1).

This gap arises from the different design goals set by GM and the Linux block layer. GM targets userspace processes which need to communicate from virtually addressed message buffers without entering the kernel, while the Linux block layer prepares requests so that it is convenient for the driver to pass them to a block device supporting DMA with physical addresses.

To solve this problem, a number of different approaches can be used:

Staging data in pre-registered buffers: The nbd client pre-allocates a number of messages buffers and registers them with GM at initialization time. All block data need to be copied in and out of these buffers, depending on the type of the I/O request: for a remote write request, data are gathered into these buffers by the CPU. For a remote read request, the Myrinet NIC matches the incoming network reply with one of the staging buffers, then the driver is notified and does CPU-based copying to scatter the blocks into their final locations.

Although this approach is simple, it has a number of disadvantages which significantly limit its applicability:

- it requires host CPU-based copying of all block data, and thus introduces significant host CPU overhead. This happens although firmware on the Lanai can use the onboard DMA engines to queue multiple DMA requests for arbitrary physical memory segments.
- it consumes three times the sustained remote I/O bandwidth on the client's memory bus; data have to cross it once during NIC-based DMA and twice during the copy operation.
- it may require modifications of the kernel's page tables in the critical path of block I/O: the nbd client is presented with a list of physical pages which are not necessarily mapped to the kernel's VM space. Depending on the architecture, only a number of physical memory pages (*low* memory, 1GB for the i386) is already mapped one-to-one to the kernel's virtual address space. The rest of memory (*high* memory) is not, and needs to be explicitly mapped before a kernel function may access it. This is not a problem when the pages are to be passed to a real block device for DMA, since only their physical addresses are needed, but for a CPU-based copy operation the driver has to map them into the kernel's VM space, and unmap them when it completes.
- Depending on the type of I/O request, page mapping and CPU-based copying happens either when the nbd client's request function is running – a write request – or before calling the block layer's completion handler – a read request. In both cases the driver locks the virtual block device and disables interrupts on the local processor, to ensure correctness. This introduces significant overhead in the critical path of block I/O and prevents

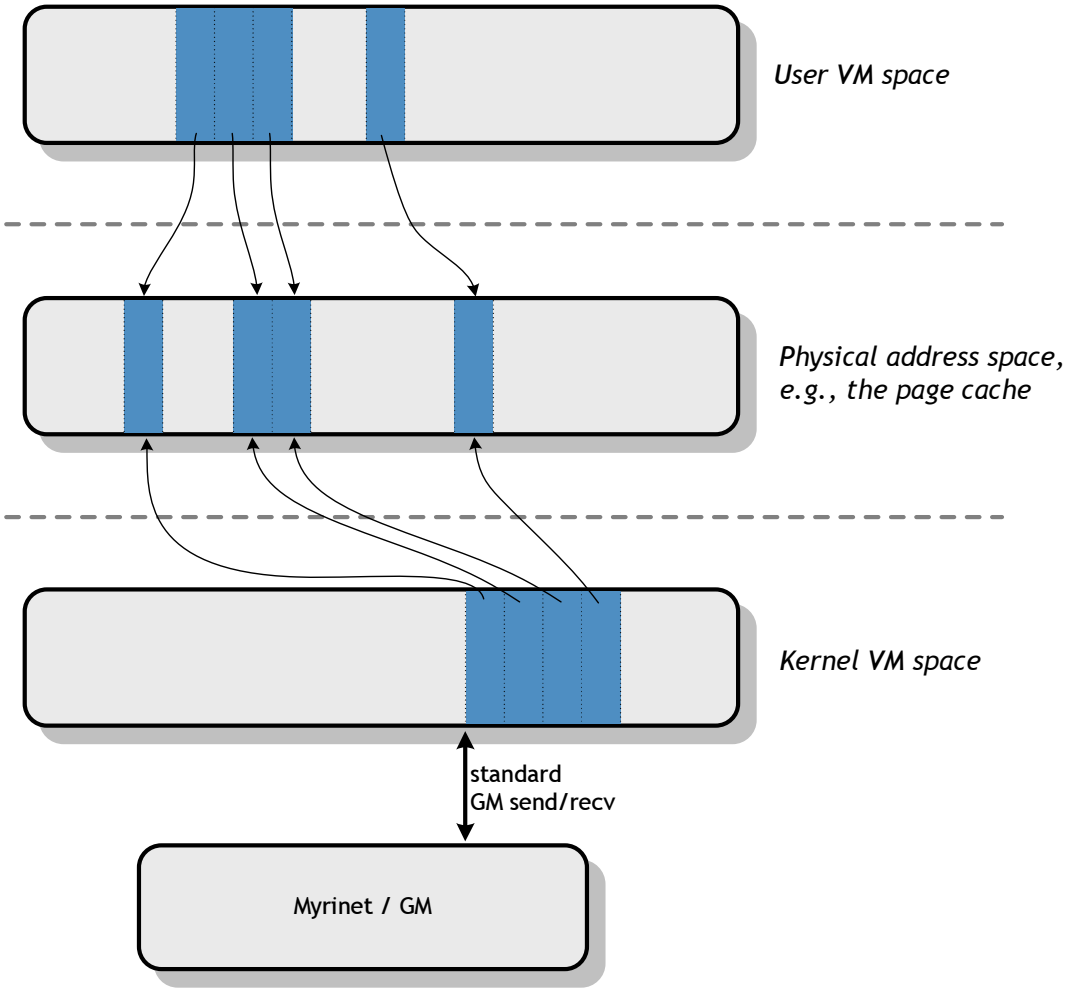


Figure 5.2: *Block I/O buffers mapped onto contiguous areas in kernel VM*

request queueing and scheduling to proceed in parallel on other processors of the system.

Mapping of disjoint physical memory pages onto contiguous kernel VM areas: To avoid the redundant data movement imposed by the previous approach, the nbd client may assemble the physical memory pages referenced by each request into a contiguous area inside the kernel's VM space (Fig. 5.2). It can then register this area with GM and use it in a network I/O operation. The area needs to be unregistered and unmapped when GM reports completion of the network operation, since each individual block I/O request refers to a possibly unrelated set of physical pages.

This approach has the advantage of being zero-copy; the Myrinet NIC traverses the virtual buffer serially and does DMA to the discontiguous physical segments,

using virtual-to-physical address translation, with no redundant memory copies.

However, the nbd client needs to establish kernel page mappings and register VM areas with GM in the critical path of *every* block operation. Registering and unregistering memory with GM is a very expensive procedure, which involves pausing the Lanai and manipulating the translation cache on the NIC. Similarly to the previous approach, this happens inside the critical path of block I/O, with interrupts disabled. The relative cost of memory copying vs. memory registration determines the request size threshold over which using VM mappings is beneficial compared to staging data in pre-allocated GM buffers.

The main problem with this approach is that it maintains mappings of referenced physical pages and involves NIC-based virtual-to-physical translation, although this is unnecessary. The need for memory registration arises when doing user level communication and the kernel cannot be invoked for virtual-to-physical address translation. However, in our case the kernel is *already* in the critical path. In fact, the caller – the Linux block layer – has taken care of pinning down the referenced physical pages if necessary (e.g., for `O_DIRECT`-based I/O with user pages) and provides the driver with a scatter-gather list of physical segments.

Using one-sided RDMA operations on the server side: The server may use one-sided RDMA-read and RDMA-write operations to fetch and store data directly into the physical memory of nbd client systems. In this approach, either the server has unrestricted access to the client's physical address space, or, to maintain system security, memory registration and deregistration calls are still used in the critical path.

All of the previous approaches impose unnecessary overhead in the operation of a kernelspace nbd client. We need a way to combine secure, two-sided GM sends and receives with physically addressable discontiguous message buffers as requested by the overlying block layer of the host OS. To this end, we propose extensions to Myrinet/GM to make it more suitable for kernel-based operation, namely the ability to perform scatter-gather I/O directly from physical memory buffers, defined in terms of scatter-gather lists (Fig. 5.3).

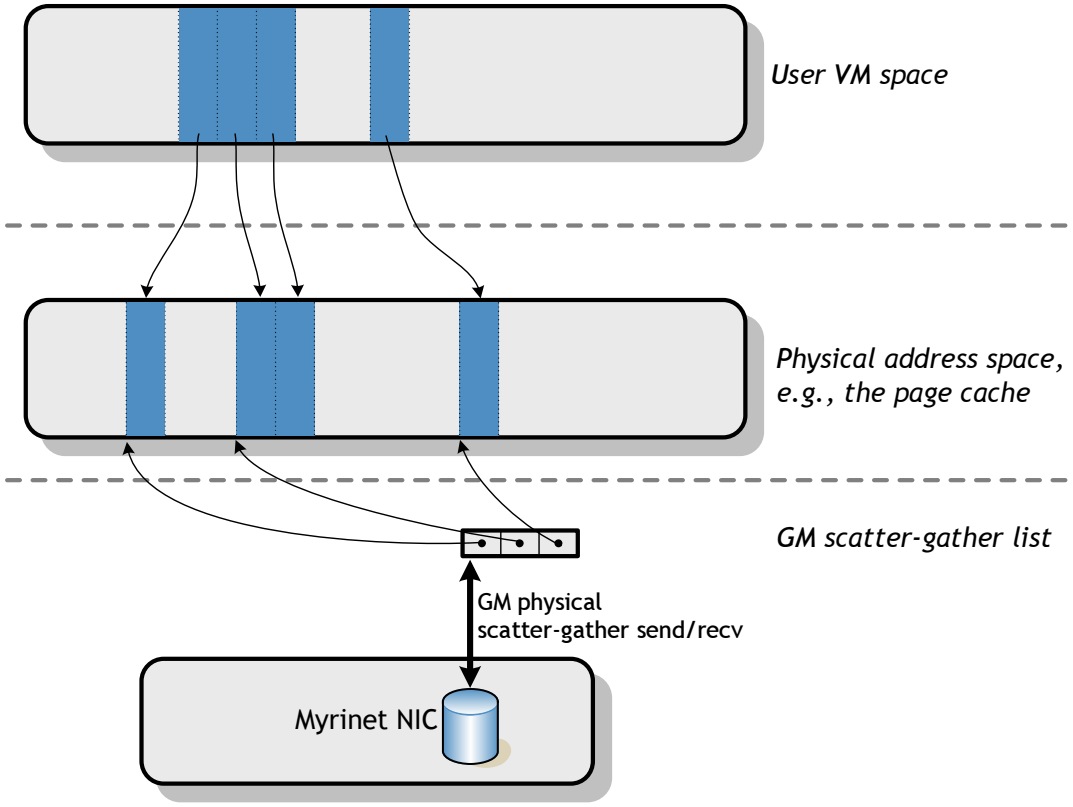


Figure 5.3: *GM physical scatter-gather I/O as a stub of the remote block device*

5.2 Proposed extensions to Myrinet/GM

5.2.1 Design

We describe the design of a proposed GM extension to support physical scatter-gather I/O. Our approach needs to co-exist both with standard GM send/receives and synchronized sends, with all kind of operations supported at the same port, simultaneously. Manipulating physical memory directly may compromise system security, so we need to ensure that the design maintains GM security semantics.

Along with standard and synchronized send operations, we define a new class of *physical gather* send operations. The message buffer involved is defined using a scatter-gather list of physical addresses passed to the Lanai. Similarly, we define *physical scatter* receive tokens.

The state of an open port is extended to include a number of GM scatter-gather lists. Every list features a number of segment vectors, i.e., {segment physical address, segment length} pairs. A scatter-gather operation happens as follows:

1. The caller initializes a free scatter-gather list entry for an open port using `gm_set_scatterlist()`.

2. The caller issues a gather send (`gm_gather_send_with_callback()`) or scatter receive request (`gm_provide_scatter_receive_buffer_with_tag()`), referring to the recently initialized scatter-gather list.

3. From this point on it the list belongs to the Lanai and may no longer be manipulated by the user, until...

4. The token is returned to the caller by GM, either when the send operation completes and the callback function executes, or when a receive event is posted to the open port's event queue.

We preserve GM's buffer matching semantics: As with standard receives, every scatter receive token also has the GM *size* attribute, and can only be matched with incoming GM packets of the same size. Receive-side buffer matching happens completely on the Myrinet NIC: the NIC traverses the scatter list before initiating a receive DMA operation.

This has three main advantages:

- Receive-side buffer matching happens completely on the Myrinet NIC. If a distinct GM *size* tag is used for every outstanding block request, the NIC has enough information to place block data directly into their final location, without any host CPU intervention.

- The CPU need only be notified when an entire network send (block write) or network receive (block read) operation has completed. All per-packet processing is undertaken by the NIC and an interrupt is raised when a whole scatter-gather list has been processed.

- The memory integrity of the client system is protected, even though no memory registration takes place. The server may request DMA writes only to the designated physical memory segments, since this is a two-sided operation and the NIC verifies the legitimacy of every network packet before starting receive DMA.

To maintain system security, only ports opened in privileged mode, i.e., by kernel entities, may manipulate scatter-gather list entries and post scatter-gather tokens to the Lanai.

5.2.2 Implementation of GM scatter-gather operations

This section describes the changes necessary to support GM scatter-gather operations in all three components of Myrinet/GM: The library, the GM kernel driver and the Lanai firmware.

GM userspace library

Functions `gm_set_scatterlist()`, `gm_gather_send_with_callback()` and `gm_provide_scatter_receive_buffer_with_tag()` are defined and exported as part of the GM library public API.

They pass gather send events and scatter host receive tokens to the Lanai, and mark them as such using flags (`GM_SEND_FLAG_PHYS_SGLIST` or `GM_RECV_FLAG_PHYS_SGLIST`).

GM kernel module

We extend the GM kernel driver to support per-port scatter-gather lists. The number of lists and number of entries per list are compile-time constants. There is a trade-off: A higher number of scatter-gather lists per-port means more block requests may be outstanding at any given time; a high number of entries per scatter-gather list means more physical segments may be coalesced in a block request to be passed over a single network message to the nbd server. However, since GM scatter-gather lists are kept in Lanai memory, this reduces the amount of memory available for caching address translations and the size of the connection array, i.e., the maximum number of nodes in the network.

GM firmware

The bulk of GM scatter-gather I/O support is implemented in the Lanai firmware. Changes affect mostly the SDMA and RDMA state machines. For every send token, the SDMA

state machine must keep track of the progress made inside the message buffer for this token. For standard tokens, this is a pair of {current virtual address, remaining length} values. For gather send tokens, the current token state is modified so that it points to the physical address space and includes information on the gather list segment currently being processed, i.e., it becomes {current physical address, current segment in gather list, total remaining length}. Whenever a new packet is to be injected into the network, data are retrieved directly from the current physical pointer, omitting any address translation. Packet length is determined by the number of bytes remaining in the current segment. When the physical pointer reaches the end of a gather list segment, it is updated to point to the following one.

The current state of the gather send token is also kept inside the send record used for implementing the go-back-N part of the network protocol. If a NACK is received, or this send record expires, the send token will be rewound to its previous state, including the current physical segment.

Similarly to the SDMA machine, the RDMA machine is enhanced, so that the scatter list is traversed before initiating receive-side DMA for scatter receive tokens. The scatter list is not accessed in ascending order, because the implementation must support the combination of synchronized sends with scattered receives: an incoming packet may refer anywhere inside the discontiguous physical buffer. Thus, whenever an incoming data packet matches with a scatter receive token, the RDMA state machine performs a binary search inside the scatter list, to find the segment where the incoming packet belongs. The final destination address for DMA is computed based on the value of the `h_synchr_ptr` field inside the GM packet header and the starting physical address of the scatter list segment.

5.3 Kernelspace gmblock client

On the client side, gmblock runs as a kernelspace driver, presenting a standard block device interface to the rest of the system. Retaining the standard block device interface makes our framework instantly usable either directly as a raw device, e.g., by VM instances or a parallel database, or indirectly, through a shared-disk filesystem using standard POSIX I/O. Moreover, we retain the highly optimized, production-quality

I/O path of the Linux kernel, which does I/O queueing, scheduling, request coalescing and mapping to physical scatter-gather lists, before presenting the requests to our driver; the performance of our nbd system benefits from improvements of the Linux block layer as its implementation evolves.

The client uses Myrinet/GM with the extensions described in the previous section for communication with the remote server. It maps every block read or write request for DMA access, then passes the resulting scatter-gather list to GM for processing.

The driver derives the limits for its block layer queue (see Section 2.3) from the compile-time parameters of the underlying GM; the number of segments in a GM scatter-gather list specifies the maximum number of segments in a block request, while the number of scatter-gather lists per port defines the maximum queue depth.

For remote read requests, receive-side matching happens entirely on the NIC, without any host CPU involvement. The driver specifies a distinct GM *size* argument per outstanding request, and a unique scatterlist for every size. Thus, any incoming block data packet matches a single scatter-gather list and the NIC can initiate a receive DMA operation independently. The host need only be interrupted when all of the block data for a given request have arrived.

Essentially, we implement a stub of the remote storage medium on the NIC itself (Fig. 5.3). This scheme, combined with the short-circuit data path on the server side, supports end-to-end zero-copy data movement from remote storage to scattered client memory segments. Scatter-gather I/O can reach all the way to user buffers for applications which implement their own data caching policies and perform direct I/O. To the best of our knowledge, this is the first such implementation.

5.4 Parallel filesystem deployment over gmblock

We completed a prototype deployment of the Oracle Cluster File System (OCFS2) over gmblock-provided shared storage, which enabled us to evaluate gmblock with real-life application I/O patterns from various workloads.

Our testbed consists of four cluster nodes in configuration A and one storage server of configuration B. All cluster nodes access the 3Ware RAID0 array over virtual devices

backed by the gmblock kernelspace client. They run OCFS2 1.5.0, which is part of the standard Linux kernel version 2.6.28.2. We had to patch the kernel for 16KB kernel stacks instead of the default 8KB, to work reliably with the long function call chains coming from stacking OCFS2 over gmblock over GM.

Using the proposed server-side data path means no server-side caching and prefetching is possible. To explore this effect, we compare two versions of gmblock: `gmblock-ramcache`, a version which passes both read and write data through RAM buffers using cached I/O, and `gmblock-sram`, which uses the proposed disk-to-NIC path for reads and only issues direct I/O requests. Writes still go through main memory, uncached, to work around the hardware limitation of the LanaiX which cannot support efficient peer-to-peer transfers as a read target (see Section 3.3).

Caches are cleared on every cluster node before every experiment to ensure consistent results.

We run three different application benchmarks on the OCFS2-over-gmblock setup: IOzone [NC], a server workload, and MPI-Tile-I/O [Ros].

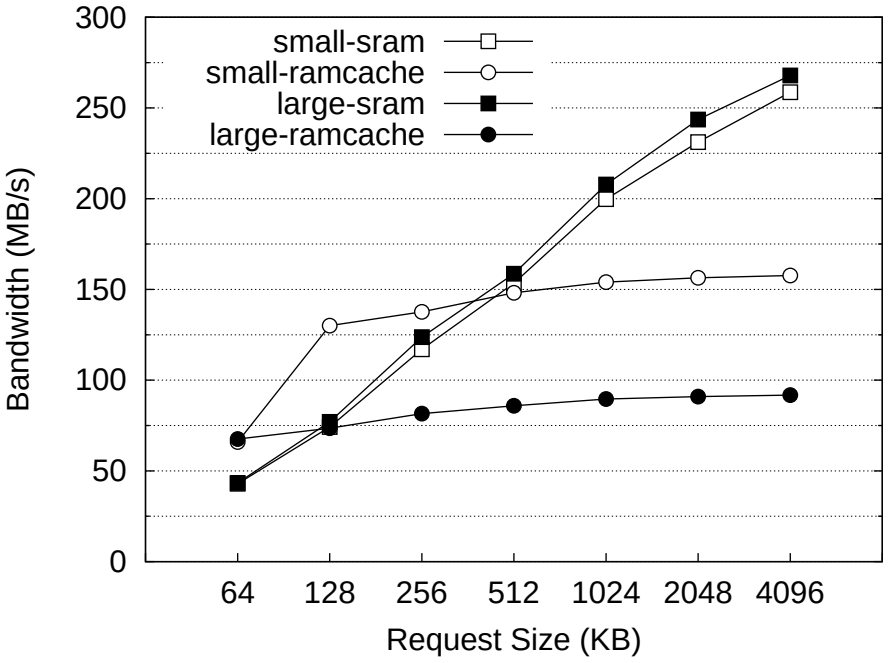


Figure 5.4: *IOzone, single-node performance: No caching, direct I/O*

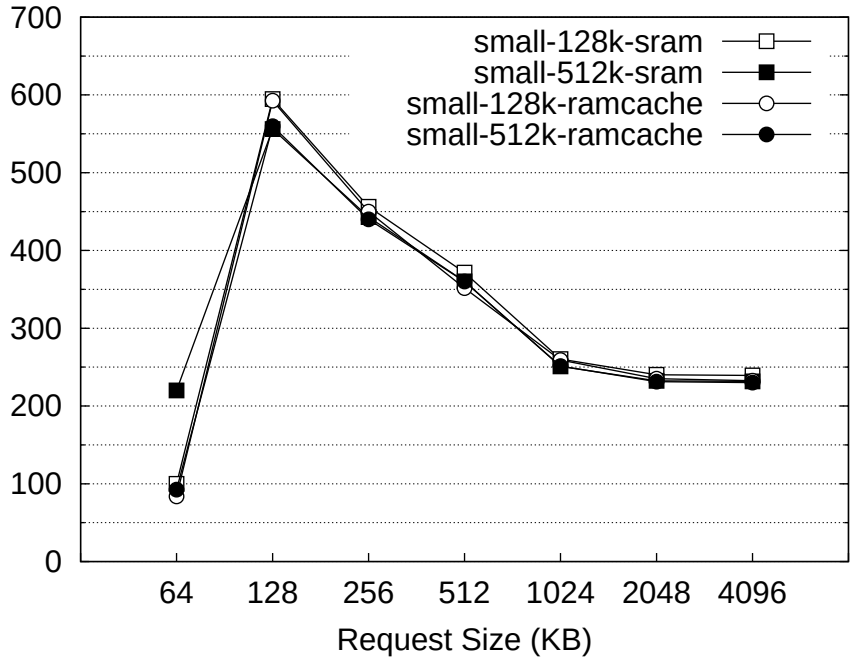


Figure 5.5: *IOzone, single-node performance: Client-side caching, small file*

5.4.1 Experiment 3a: Single-node IOzone performance

IOzone is a filesystem benchmark generating a variety of different I/O patterns. We tested its performance in the read, re-read, and write modes. For every test, IOzone performs multiple passes varying the I/O size from 64KB to 4096KB. We used two different workloads. A "small" file of 512MB which fits entirely in a node's cache and a "large" file of 4GB. This is to demonstrate server and client-side cache effects. We show results from the read tests, since the re-read case coincides with the small file read case after the first pass, and writes follow the same data path in both implementations.

We look into the base performance of IOzone with a single client. Fig. 5.4 shows the performance of uncached (direct I/O) reads for various request sizes both for `gmblock-sram` and for `gmblock-ramcache`. We see that `gmblock-ramcache` is capped by the memory-to-PCI bandwidth at ∼93MB/s and ∼160MB/s for the large and small file respectively. The small file case has considerably better performance because the first 64KB pass brings the entire file in storage server RAM. Thus there is no disk-to-memory traffic competing with cache-to-NIC traffic.

The `gmblock-sram` case scales linearly with request size independently of file size since there is no client or server-side caching. It is interesting to note that `gmblock-ram`

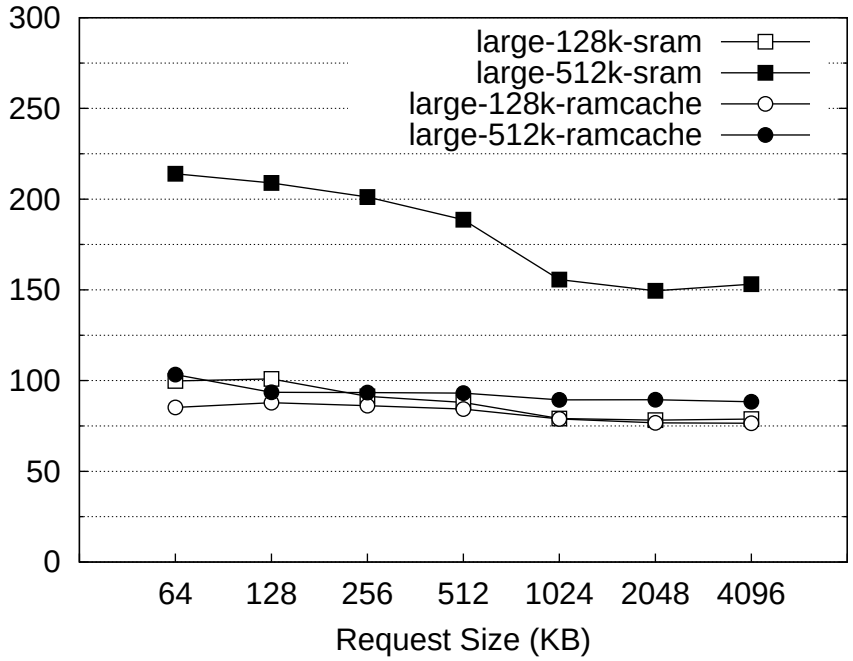


Figure 5.6: *IOzone, single-node performance: Client-side caching, large file*

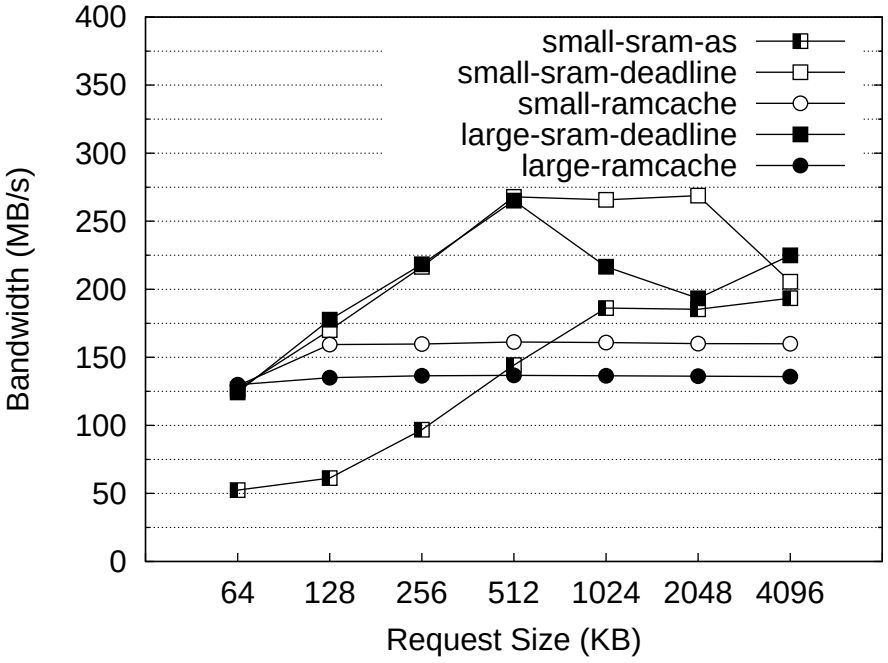


Figure 5.7: *IOzone, multiple-node performance: No caching, direct I/O*

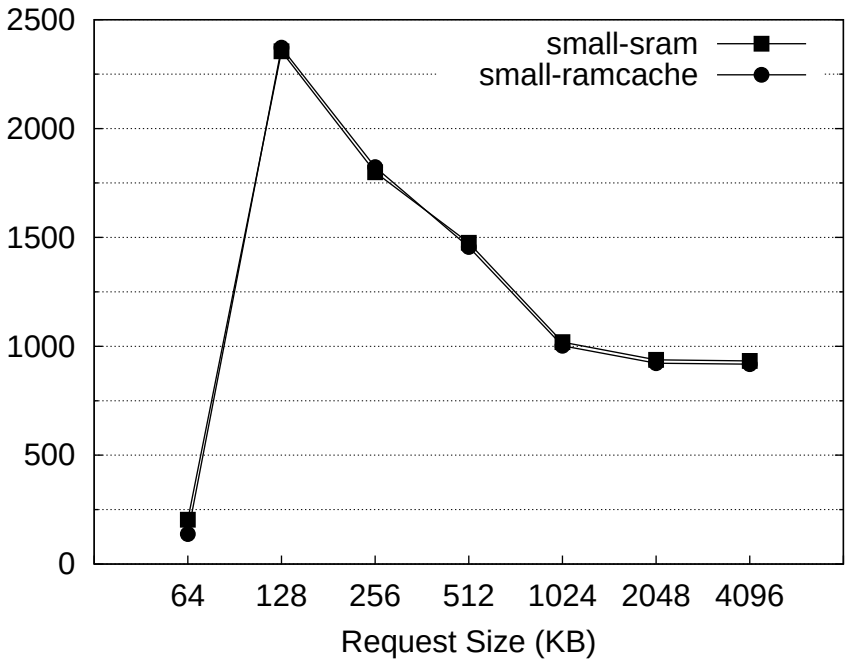


Figure 5.8: *IOzone, multiple-node performance: Client-side caching, small file*

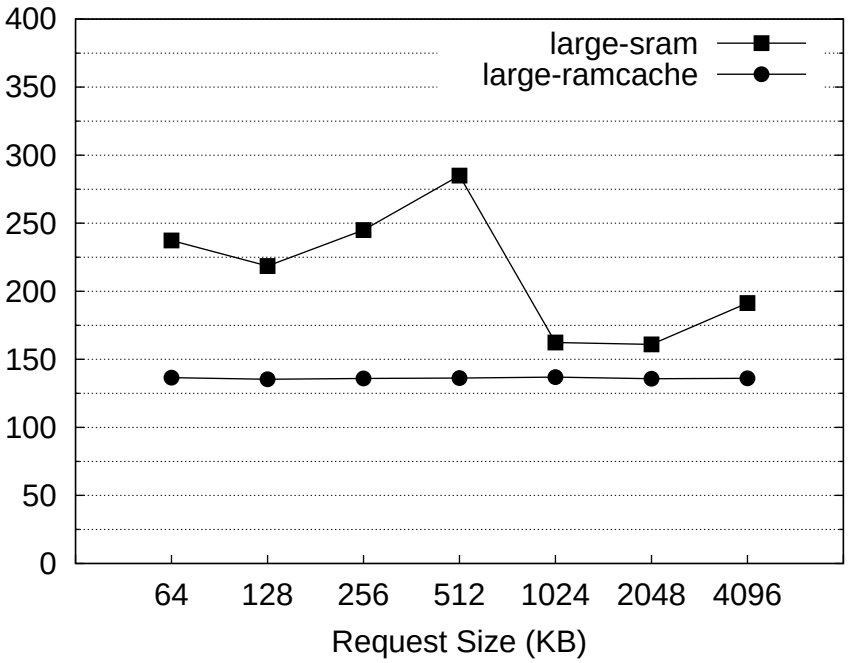


Figure 5.9: *IOzone, multiple-node performance: Client-side caching, large file*

outperforms `gmblock-sram` for the initial read of 64KB requests. This is because the caching server may prefetch aggressively: the caching server takes advantage of readahead set at 512KB, so `gmblock-sram` begins to outperform `gmblock-ramcache` after the 512KB request mark and is ∼1.64 and ∼2.9 times better for the small and large file respectively.

Fig. 5.5 shows small file performance, for readahead settings of 128KB and 512KB. We see client-side caching in effect. For request sizes after 128KB, I/O requests are being served by the page cache on the client side, as every cluster node gets its own read lock on the shared data and caches the file independently. The steep drop as the request size increases is an artifact of processor cache behavior. As request size increases towards 512KB, IOzone's application buffer no longer fits in the L2 cache of the CPU, so RAM becomes the target of memory copy operations when hitting the local page cache. With a large enough request size, we essentially see memory copy bandwidth limitations. `gmblock-sram`'s behavior for the initial file read shows that prefetching is necessary to achieve good performance. When using the direct disk-to-NIC data path it is not possible to prefetch on the server, but client-initiated prefetching is still possible. The initial file read with 512KB readahead on the client outperforms the 128KB readahead setting by as much as 120%. Thus for all remaining experiments we continue with a readahead setting of 512KB on the clients.

Finally, Fig. 5.6 shows read performance for the large file case. This time, no cache reuse is possible. Performance of `gmblock-sram` with 128KB readahead is low because the application I/O request must complete fully before a new one can be issued by IOzone. When readahead is set 512KB the Linux I/O layer will overlap data prefetching with page cache to application buffer copying. The bandwidth drop after 512KB is an artifact of L2 caching as in the small file case.

5.4.2 Experiment 3b: Multiple-node IOzone performance

We repeat the previous experiments, this time with four instances of IOzone running concurrently, one on each client node. Fig 5.7 demonstrates the aggregate attained bandwidth for direct I/O reads of various request sizes. All IOzone processes work on the same 512MB or 4GB file. This is the best possible scenario for `gmblock-ramcache`; It has consistently good performance, since it only fetches data once in memory, then

all nodes can benefit from it as they read through the file at approximately the same rate. Achieving good performance with `gmblock-sram` proved more difficult. It quickly became apparent that the choice of the I/O scheduler on the server was crucial. Initial testing with the anticipatory scheduler showed poor disk efficiency. The bottleneck was the storage medium serving a seek workload. Testing with other schedulers available in the Linux kernel (deadline, noop, CFQ; only presenting results for deadline, for brevity) showed that performance varied significantly with request size. In the best case, I/O scheduling increases disk efficiency enough for `gmblock-sram` to outperform `gmblock-ramcache` by 66%. In the worst case, `gmblock-sram` only achieves 40% of `gmblock-ramcache`'s performance, for 64KB requests and the anticipatory scheduler.

Fig. 5.8 shows small file performance. After the initial read of the file, all clients read from the local caches concurrently at a rate of ∼2.4GB/s.

Fig. 5.9 shows large file performance for 512KB readahead. `gmblock-sram` consistently outperforms `gmblock-ramcache`, but its performance is very sensitive to the application request size. We attribute this to the interaction of request timing with server-side I/O scheduling.

5.4.3 Experiment 3c: Server workload

We evaluate a web farm scenario, where all four clients run scripts simulating web server instances. Each instance serves randomly chosen files of fixed size from a single directory in the shared filesystem. We use the number of files served in a 2-minute period as a metric of sustained system throughput. The file set ranges from a small cacheable workload (70 files of 10 MBs each), to 1000 and 10000 files of 10MBs each (Fig. 5.10(a)).

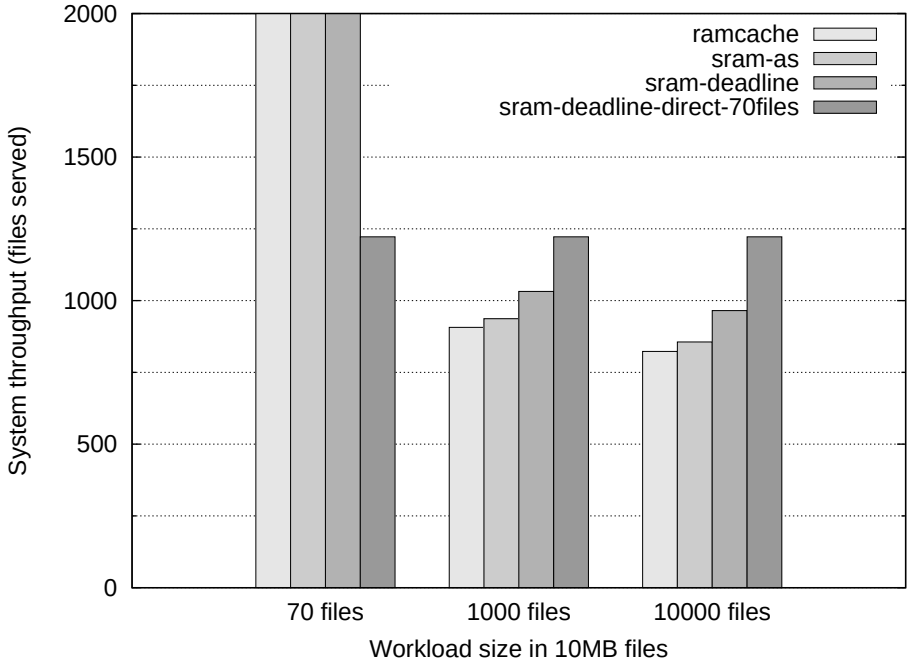
For the cacheable workload, there was no significant difference in the performance of `gmblock-ramcache` and `gmblock-sram` (results reached ∼7000files/2min and are off the chart). All clients quickly built a copy of the workload in their page caches, so the result is dominated by the memory copy rate when hitting the page cache. `gmblock-sram` performs 12% and 17% better for the 1000 and 10000-file case, respectively. It is bound by disk performance due to small file seeks. To confirm this, we repeat the test with the small 70-file workload, this time reading in `O_DIRECT` mode, to prevent any client-side caching (bar "sram-deadline-direct-70files" in the chart). With the disks

performing seeks in a narrower range, performance improved by an extra 18% and 26% compared to the 1000 and 10000-file case respectively.

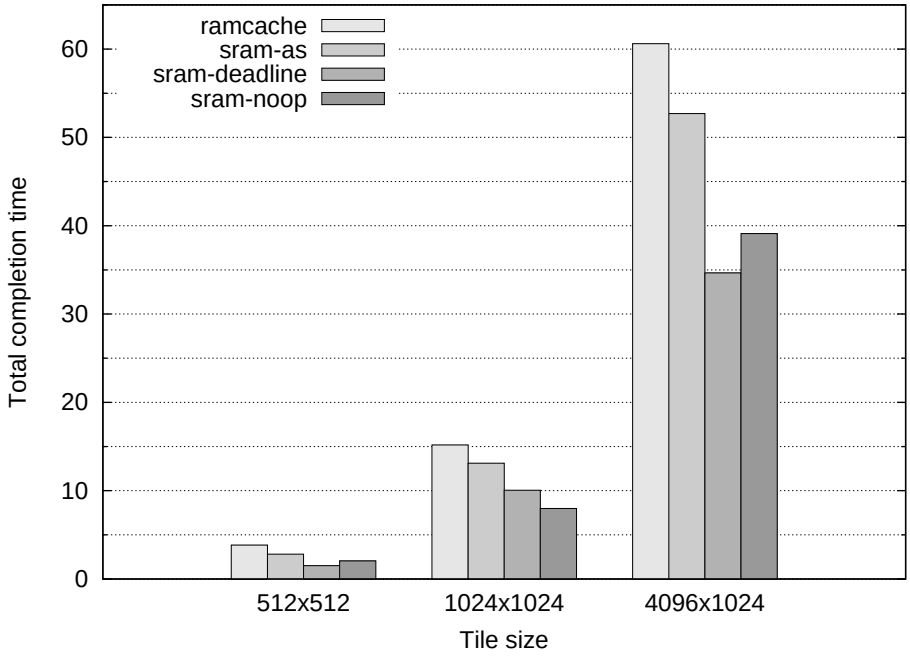
5.4.4 Experiment 3d: MPI-IO Application

We use MPI-Tile-IO, an MPI-IO benchmark which produces a non-contiguous access workload similar to that of some visualization and numerical applications. The input file for the application is divided in a dense 2D set of tiles, with each peer process accessing a single tile. We perform the I/O needed to render a frame on a 4×4 tiled display, with 512×512, 1024×1024, or 4096×1024 tiles and 32 bytes per element. Total completion time for every configuration is shown in Fig. 5.10(b). In the best case, `gmblock-sram` delivers 39%, 51% and 57% the completion time of `gmblock-ramcache` for the three tile sizes respectively, although no I/O scheduler has consistently better performance.

Overall, `gmblock-sram` performed better for all three workloads. However, the effect of the short-circuit data path only becomes visible when server-side I/O scheduling can remove the disk bottleneck due to concurrent access. This is why the performance increase is more pronounced for IOzone, whose access pattern comprises multiple peers streaming data concurrently, compared to the server and scientific application workloads, which lead to more frequent seeks.



(a) Web server workload



(b) MPI-Tile-I/O performance

Figure 5.10: *Web server workload and MPI-Tile-I/O performance*

6 Related work

This work describes a framework for block I/O over intelligent interconnects with direct disk-to-NIC data transfers; the aim is to minimize host CPU involvement and to mitigate the impact of resource contention on the I/O path of commodity storage servers.

Thus, it is relevant to past and ongoing research on user level architectures, efficient utilization of shared resources on clusters of SMPs, shared block storage systems, parallel and distributed filesystems, and novel interconnection technologies with programmable network interfaces. In the following, we present related work in each of these areas.

User level networking

Myrinet/GM is used as the message-passing networking substrate for gmblock. The main concepts of user level networking, a number of significant architectural choices for a ULN system and the position of Myrinet/GM in the resultant design space, have been discussed extensively in Section 2.2.

Efficient utilization of shared resources on SMPs

Substantial work has focused on the problem of limited bandwidth of shared resources on SMP systems. The work in [LVE00] targets the impact of memory bus contention on multiprocessor systems running a mix of non-realtime and realtime applications. The authors emphasize that slowdown due to memory contention is possible even when the total memory bandwidth demands of applications do not exceed the available memory bandwidth, then explore scheduling strategies which aim to fulfill memory band-

width guarantees for high-priority processes. Bellosa [Bel97] explores the impact of time-sharing applications executing concurrently with real-time applications on different processors of a multiprocessor system. He proposes techniques for throttling lower priority applications to reduce the load on the memory bus and mitigate the effects of memory contention. Schönberg [Sch03] focuses on the effect of I/O load on the total execution time of applications due to memory load induced by movement of data between peripheral devices and main memory. He describes metrics for quantifying application slowdown, which can be used to amend scheduling decisions when aiming to meet resource reservation requirements. The work in [ANP03] describes a system which aims to coordinate the execution of applications for reduced memory contention, by scheduling high-bandwidth demanding applications with low-bandwidth demanding applications. The system is based on source code changes to track the memory bandwidth usage of processes based on hardware performance counters [Pet04], but does not take into account the memory bandwidth consumption of applications due to network and disk I/O. Finally, [WS06] describes a scheduling framework which aims to reduce contention for shared resources on a multiprocessor by selecting suitable applications for space sharing based on user-provided information.

Impact of data movement in commodity storage servers

The problem of redundant data copying and the overhead of data movement in the I/O path of storage systems has been explored in various research efforts. The work in [PSC03, PSC05] explores the performance of servers for network-attached storage. The authors observe the operation of NFS servers backed by iSCSI storage and highlight that it is dominated by data movement, rather than data processing, which is minimal. To avoid redundant data copying while serving data, they propose changes to the organization of the buffer cache, so that data brought from storage are kept in a network-ready format in main memory The layers of the network stack are enhanced so that instead of copying the actual data in main memory, they exchange references to them. Our work shares the same premise, that the overhead of server-side data movement is pivotal to performance, while data interpretation on the server is less important. We propose device-to-device communication as a means to alleviate storage server load, aiming to bring data from server storage to their final location in the buffer cache of the client,

with minimum overhead. A unified I/O buffering and caching system is presented in [PDZ00], while a similar approach for unified I/O and communication buffer management in PVFS storage servers is explored in [WWPR04]. Therein, the authors emphasize the overhead of data movement as the available bandwidth for RDMA-capable interconnects becomes comparable to main memory bandwidth.

Shared-disk filesystems

A large fraction of the storage infrastructure for medium and large-scale clustered systems is based on shared-disk filesystems. Such systems include IBM's GPFS [SH02], Oracle's OCFS2 [Fas06],Red Hat's Global File System (GFS) [SRO96, PBB+99], SGI's CXFS [SE04], an extension of XFS [SDH+96] to support clustering, and the VERITAS Cluster File System [Sym]. To ensure consistency while accessing shared storage, shared-disk systems commonly employ a distributed lock manager (DLM) based on the principles and interface of the VMS DLM [KLS86]. These filesystems typically operate over physically shared storage, e.g., over Fibre Channel. They can also be deployed over a *virtual* shared storage pool, as provided by an nbd system. Section 5.4 describes the performance of an OCFS2 installation over gmblock.

Scalable clustered storage

Much research has focused on provided flexible distributed clustered storage by combining storage from various nodes in a virtual shared disk infrastructure. Frangipani [TML97] is a shared-disk clustered filesystem operating on top of a shared storage pool provided by a Petal virtual disk [LT96]. The work in [FLB08] presents Orchestra, a system to form virtual storage hierarchies by building on the Violin [FB05] block I/O framework. It replaces the traditional block I/O interface with a richer interface which enables in-band support of both data and control requests, for operations such as locking and space allocation. While this provides for more flexibility at the block layer, it requires rewriting of the overlying filesystem layer to be usable with Orchestra.

Object-based Storage Devices (OSDs) have also emerged as a major research trend on scalable storage. An OSD is a storage device which uses *object* structures rather than blocks, as the fundamental unit of data storage; an object being the combination of file

data plus a set of associated attributes [osd04]. OSD capabilities may be implemented at various levels of the hierarchy, e.g., the storage server or on the actual disks, by including increased processing capability close to storage [GNA+98]. Lustre [Sun08] is a widely used cluster filesystem based on object storage. It is based on decentralized processing of metadata and data; metadata are handled by Metadata Servers (MDS), while data are handled by Object-Storage Servers (OSS), stored on Object-Storage Targets (OSTs), which currently are modified `ext3` filesystems. Our work on gmblock can be applied to designing a more efficient OSS; the traversal of `ext3` structures for OST management can run on the CPU with minimal interference from block transfers between the storage device and the network. Similarly, the proposed data path can improve the scalability of embedded systems which are combined with commodity disks to provide object-storage capabilities, as is the Panasas StorageBlade [NSM04, WUA+08].

Traditionally, the Network File System (NFS) has been used to provide remote access capabilities in UNIX systems. Its latest revision, NFSv4.1 includes Parallel NFS (pNFS) support [HH05]. To support I/O from a single client to multiple storage servers, pNFS separates file data from file metadata and moves the metadata server out of the data path. A client initially interacts with a single meta-data server, which provides it with a *data layout*. The client uses the data layout as a template to contact the storage servers directly and perform data updates over multiple, parallel paths. The data layout may specify various access protocols; the standard supports block-based storage, Object-Based storage, or file-based access to the data chunks. Our framework can be used as one more means of block-based access to storage servers, to complement an existing pNFS installation. Uncoupling data from metadata handling enables industry-standard NFS clients to interact with gmblock-enabled storage with no code modifications.

Network block devices

TCP/IP-based approaches for building nbd systems are well-tested, widely used in production environments and highly portable on top of different interconnection technologies, as they rely on – almost ubiquitous – TCP/IP support. They include the Linux Network Block Device (NBD), Redhat's Global NBD (GNBD) used in conjunction with GFS [PBB+99], the Distributed RAID Block Device (DRBD) [Ell07] and the GPFS Network Shared Disk (NSD) layer [SH02]. On the other hand, they exhibit poor perfor-

mance, need multiple copies per block being transferred, and thus lead to high CPU utilization due to I/O load. Moreover, using TCP/IP means they cannot access the rich semantics of modern cluster interconnects and cannot exploit their advanced characteristics, e.g., RDMA, since there is no easy way to map such functions to the programming semantics of TCP/IP. As a result, they achieve low I/O bandwidth and incur high latency.

RDMA-based implementations [KKJ02, LPB04, LYP06] relieve the CPU from network protocol processing, by using the DMA engines and embedded microprocessors on NICs. By removing the TCP/IP stack from the critical path, it is possible to minimize the number of data copies required. However, they still feature an unoptimized data path, by using intermediate data buffers held in main memory and having block data cross the peripheral bus twice per request. This increases contention for access to main memory and leads to I/O operations interfering with memory accesses by the CPUs, leading to reduced performance for memory-intensive parallel applications.

The work in [MXPB06, MPB07] addresses the end-to-end performance of a a Linux 2.4 kernel-based block sharing system, over a custom 10Gbps RDMA-capable interconnect with exclusive access. The authors show that network and disk interrupt processing overhead can have major impact on the attainable performance and focus on I/O protocol optimizations in order to alleviate it. We follow a different approach, focusing on integration in an existing HPC infrastructure: gmblock is implemented in userspace, in order to simplify its design and be able to access structured storage (e.g., data in a filesystem) using kernel-provided abstractions. To mitigate the overhead of managing disk and network events we propose synchronized send operations, so that servicing of larger requests can happen with coordination between storage and the network, without host CPU involvement.

Exploitation of memory onboard the Network Interface

The availability of NIC-based memory has spawned research efforts [KPR02, yKRP05, CKE+05, WYMG09], which seek to increase server efficiency by engaging it for on-NIC caching of data. In [KPR02], the authors enhance FreeBSD's `sendfile()` system call to support NIC-based caching on a programmable Gigabit Ethernet NIC. They evaluate the performance of a web serving scenario with various amounts (up to 16MB)

of emulated NIC memory. In [CKE+05], experimental evaluation of a cluster-based web server interconnected with Myrinet shows NIC-based caching reduces server load significantly, by reducing PCI bus contention and avoiding the latency of DMA operations for inter-node communication. Similarly, [WYMG09] notes that network bandwidth is increasing to become a significant fraction of the peripheral bus bandwidth, and proposes a hierarchical cache architecture with on-NIC read caching for iSCSI storage servers.

Device-to-device data movement

Our framework is not the only one to support a direct path between storage media and the network; a number of network block sharing systems have been described in the literature supporting block transfers from the disk to the NIC. However, they impose limitations which can limit their applicability in real-world scenarios.

The DREAD project [Dyd01] describes a mechanism of controlling SCSI storage devices over SCI-provided remote memory accesses to their PCI memory-mapped I/O space; in this setup, the SCI controller driver runs on the client. However, only a single remote node may be running the driver and accessing the SCSI adapter, hence no data sharing between multiple clients is possible. Also, the system needs source code changes to the SCSI driver so that it performs I/O over SCI-mapped memory. This approach has significant interrupt overhead; interrupts are routed via the server CPU, causing a network transaction and an interrupt on the client, which is caught by DREAD and routed to the SCSI driver. Thus, it does not scale well as the I/O rate increases. Finally, there is little room for client- or server-side optimizations to the protocol, since it works at a very low level directly between the SCSI driver and the device, as if they were directly connected over the PCI bus.

The work on Proboscis [Han01, HL02] implements a block-level data sharing system over SCI. It builds on the idea of having nodes act as both compute and storage servers and describes a kernel-based system for exporting block devices. It also mentions the possibility of direct disk-to-NIC transfers, by exploiting hardware support specific to SCI for mapping remote memory to a node's physical address space. This reduces host overhead significantly, but may be problematic: first, it would only make sense for disk read operations (remote memory writes), since the overhead of remote reads over SCI

is prohibitive; second, there is a low limit on the number of SCI memory mappings that may be active at any time, which would interfere with processes trying to make concurrent use of the interconnect – a problem analogous to Myrinet's limited amount of SRAM on the NIC; third, referring to SCI-mapped addresses directly makes error handling in case of network failures very complicated, as there is no way for the storage device to be notified whenever a memory access operation to a physically mapped remote location fails; and finally, there is no provision for coherence with the local OS's page cache. Our framework proposes synchronized sends to work around the limited amount of SRAM on the Myrinet NIC and avoids the problem of error handling by isolating network I/O in a distinct SRAM-to-wire or wire-to-SRAM phase.

The work on Off-Processor I/O with Myrinet (OPIOM) [Geo02] was the first Myrinet-based implementation of direct data transfers from a local storage medium to the NIC. At the server side, OPIOM performs read-only direct disk-to-Myrinet transfers, bypassing the memory bus and the CPU. However, to achieve this OPIOM makes extensive modifications to the SCSI stack inside the Linux kernel, in order to intercept block read requests so that the data end up not in RAM but in Lanai memory. This has a number of significant drawbacks: first, since the OPIOM server uses low-level OPIOM-specific SCSI calls to make such transfers, it can only be used with a single SCSI disk. Moreover, there is no provision for concurrent accesses to the SCSI disk, both over Myrinet and via the page cache, thus no write support is possible, since there is no way to invalidate blocks which have already been cached in main memory, when a remote node modifies them. Even for the read case, it is unclear what would happen if a remote node requested data recently changed by a local process, still kept in the page cache but not yet flushed to disk. Our proposed framework is able to ensure coherence with the page cache, both for reads and for writes by exploiting the direct-I/O semantics of the Linux 2.6 kernel; it will invalidate cached blocks before an `O_DIRECT` write and will write back any relevant cached pages to disk before an `O_DIRECT` read. By integrating parts on Lanai SRAM into the VM infrastructure provided by Linux and using `O_DIRECT`-type transfers, gmblock is disk-type agnostic and can construct an efficient disk-to-network path regardless of the underlying storage infrastructure, whether it is an IDE disk, a SATA disk, storage accessible over Fibre Channel or even a software RAID0 device. The only requirement is that a Linux driver capable of using DMA to service `O_DIRECT` transfers is available.

A different approach for bringing storage media closer to the cluster interconnect is

READ^2 [CRU03]. In READ^2, the whole of the storage controller driver resides on the Lanai processor itself, rather than in the Linux kernel. Whenever a request arrives from the network, it is processed by the Lanai, which is responsible for driving the storage hardware directly, thus bypassing the host CPU. However, removing the host CPU (and thus the Linux kernel executing on top of it) from the processing loop completely, limits the applicability of this approach to real-world scenarios for a number of reasons. First, it disregards all the work devoted to developing stable in-kernel block device drivers; for each different block device, its driver needs to be rewritten to run on the limited resources of the Myrinet NIC, which provides none of the hardware abstraction layers of the Linux kernel. Second, since one cannot have two different agents driving the same storage controller without any coordination, the disk being shared is inaccessible by the host system. As a result, no enforcement of per-user rights and no process isolation is possible. Finally, even for short commands to the storage controller, the Lanai cannot do PIO, but is instead forced to setup DMA transactions from and to the controller's I/O space. Thus, the latency of control operations becomes very high. The design of gmblock does not seek to completely decouple the host CPU and host OS kernel from network processing. Instead, it involves them at points where it is beneficial, i.e., during the block transfer setup phase. The host CPU, with its very high clock frequency, is much better suited to program the storage controller with PIO than the Lanai. Moreover, the kernel provides all the different abstractions and facilities (block device layer, device drivers, virtual memory subsystem, different privilege levels and more) which are necessary for maintaining code portability, process isolation and memory protection.

Client-side optimizations

Regarding the client-side operation of our nbd system, a number of research efforts have explored the performance of accessing distributed and parallel filesystems over user level interconnects and the imposed overhead, which is mostly due to the need for noncontiguous memory buffers in virtual and physical memory and the need for memory pinning.

The work in [WWP03, YP05] proposes client-side optimizations for userspace-based access to parallel filesystems. PVFS2 over Infiniband and PVFS2 over Quadrics are used as case studies, respectively. The focus in [WWP03] is on matching the semantics of

PVFS non-contiguous I/O operations and the PVFS List-IO interface [CCkL+02] with the capabilities of modern interconnects for zero-copy I/O. The work exploits RDMA scatter/gather for noncontiguous access and proposes mechanisms to reduce the overhead of Infiniband memory registration. Our approach employs a kernelspace block driver, thus the kernel undertakes memory pinning and virtual-to-physical translation, and the need for memory registration is eliminated.

The work on ORFA [GP04, GPG04] examines the client-side overhead of accessing a distributed filesystem from userspace, over Myrinet. The authors begin with a userspace implementation and study the performance of a pin-down cache which enables it to use of GM's user level networking facilities directly. They port their implementation to kernelspace, to take advantage of kernel-provided client-side caching and encounter problems similar to those described in Section 5.1.2 with regard to discontiguous physical memory buffers due to application usage of VM buffers. They propose extending GM to support physical addresses directly, since the kernel is in the critical path. However, support is limited to contiguous physical memory, which limits the applicability of this approach when transferring block data into the physically discontiguous page cache, and prevents aggressive coalescing of I/O requests by the client-side I/O scheduler. Our proposed extensions support sender- and receive-side I/O to discontiguous memory segments, in a single GM operation, with full NIC-based matching. Moreover, the receive path supports random fragment placement during scatter list traversal, to accommodate GM synchronized sends.

7 Conclusions and future directions

SMP systems are commonly used as building blocks for scalable clustered platforms. Resource sharing, which is inherent in such systems, can have significant performance impact. For data-intensive workloads, as the number of cores and consequentially the processing power per processor increases, the performance of the I/O subsystem becomes decisive to overall performance.

Based on these observations, this work focused on efficient, low-overhead data transport mechanisms from mass storage devices to processing cores over the interconnection network, aiming for reduced CPU, memory and peripheral bus bandwidth pressure. To this end, we employ features provided by modern DMA- and processor- enabled cluster interconnects.

To mitigate the impact of remote I/O on commodity storage servers, we proposed direct data paths between storage media and the network, bypassing the host CPU and memory bus. We presented gmblock, an nbd system over Myrinet which enables building such paths in a block-device independent manner, combining NIC-based memory with the direct I/O capabilities of the host OS. Experimental evaluation of a prototype implementation demonstrated that it delivered significant bandwidth improvement for remote I/O, with minimal interference with computation on the local CPUs.

We proposed minor hardware modifications, namely the inclusion of small, fast memory areas close to the NIC, to make our approach applicable to interconnection technologies other than Myrinet. We argue that our approach enables the use of aggressive power-saving techniques on low-power, low-frequency embedded storage servers, by detaching the control and data paths and moving the bulk of data transfer off main

memory.

We proposed synchronized send operations to enable processing of large I/O requests on the direct disk-to-NIC path, with minimal host CPU involvement. Their operation allows for intra-request overlapping of disk- and network I/O, with multiple data streams from RAID storage.

On the client side, we showed how exploiting NIC programmability to implement a stub of the remote storage device enables end-to-end zero-copy data paths, directly from remote storage to userspace buffers dispersed in physical memory. To the best of our knowledge, this is the first such implementation.

Deployment of a parallel filesystem on top of gmblock showed gmblock can mitigate the effect of resource contention and improve the performance of various real-life workloads, provided server-side I/O scheduling can sustain sufficiently high disk I/O performance with multiple data streams.

In the following we discuss some directions for possible future extensions of this work.

Recent work in the field of storage servers [PFB09] has focused on I/O scheduling techniques for multiple concurrent sequential data streams. Such techniques could be used in conjunction with gmblock to improve disk efficiency, however they would need to be adapted to work with limited on-NIC memory and to cache data on the client side.

Our work on gmblock has also highlighted a longer-term research goal; our view is that part of the problem of limited memory and peripheral bus bandwidth arises from a *semantic* gap between the host system OS and the actual underlying architecture. The advent of modern cluster interconnects has brought programmable microprocessors and fast memories residing far from the host CPU, on the peripheral bus, close to the network. However, the OS design keeps a host-RAM-centric approach with regard to I/O: All I/O operations, both at the kernel and at the user level, involve some sort of "memory buffer", or "staging area", which is assumed to lie in host RAM. The design of gmblock enables the use of memory onboard a peripheral device as the intermediate buffer, thereby constructing the direct disk-to-NIC data path.

What is needed is for the OS to be made *aware* of this new situation. Our current approach constructs pageframes in the Linux VM subsystem for Myrinet SRAM areas, but they are marked as reserved. Their use is to make direct-I/O operations possible, but

the kernel is not responsible for managing them. A goal for future work is to enhance the kernel so that it actually treats them as a separate memory space. This way, we may restore the page cache in its place inside the I/O processing path; it will be distributed between host RAM and PCI-mapped memory spaces provided by storage and network devices.

The kernel's allocation semantics can then be enriched so that an application, e.g., our userspace nbd server, may provide *hints* as to what kind of usage can be expected from a buffer to be allocated, e.g., specifying that a file or block device is to be mapped *close to the network*. Moreover, the semantics of managing the page cache and the `struct page` abstraction will need to be extended beyond the host CPU(s). Accessing and moving pages will vary significantly depending on their position and copying pages may no longer be the sole responsibility of the host CPU(s); DMA or copy engines closer to their location will be able to do it much more efficiently.

At the architectural level, this can be made possible by having the devices export on-board physical memory as PCI-mapped memory resources. This way, our Myrinet-based implementation could have the Lanai DMAing data directly into the cache of the RAID controller so that they may referred to in a future write operation, without any modifications to the controller's host driver.

From a practical standpoint, one way to reach a prototype implementation of the system being described is by using programmable NICs offering a separate PCI-X slot for attaching other storage and network devices, e.g., Intel IOP-based intelligent NICs. The combination of DRAM on the intelligent NIC with a Myrinet adapter or a storage controller, will provide the necessary functionality of having globally addressable memory spaces close to the network or the storage medium.

Extending the Linux page cache mechanism so that it may span multiple, disjoint memory areas on different parts of a computing system can bring a number of significant advantages:

- There are two distinct phases in servicing network block I/O requests: the first comprises block data transfers between storage and NI memory, the second comprises block data transfers between NI memory and the network link. To construct a direct path from/to storage, our current approach intertwines these two

processes. Treating NI memory as fully-fledged memory in the VM subsystem would enable the two processes to progress asynchronously with regard to each other.

This improves the semantics of communicating with the kernel; it is now aware of and may manage all of the available memory and use it to optimize block I/O with the storage device. By uncoupling network I/O from local I/O, latency-hiding techniques become possible, with the kernel reading ahead storage blocks into pages yet untouched by network requests. Still, this happens over the short-circuit data path. Similarly, destaging of data from NI memory to disk happens asynchronously, with the kernel performing page flush operations as the available NI memory decreases.

- With direct I/O, server-side readahead is disabled, since the kernel is only aware of the buffer involved in each individual I/O request. Restoring the page cache enables the kernel to read ahead on the server side, while still moving data directly over the peripheral bus.

- The system can support a hierarchical organization of available memory spaces, without any specific code provisions. When memory on the NI does not suffice, the system may degrade gracefully to using buffer spaces off the NIC. Initially it may use buffers close to the NIC. Afterwards it may start to allocate pages residing in host RAM. There is a tradeoff: we gain more cache space and are able to hold a larger working set of blocks in memory, in exchange for *gradual* increase in memory pressure on the host.

Bibliography

[ACMM07] Nevine AbouGhazaleh, Bruce R. Childers, Daniel Mossé, and Rami G. Melhem, *Near-memory Caching for Improved Energy Consumption*, IEEE Trans. Comput. **56** (2007), no. 11, 1441–1455.

[ANP03] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou, *Scheduling Algorithms with Bus Bandwidth Considerations for SMPs*, Proceedings of the 2003 International Conference on Parallel Processing (ICPP 2003), Oct 2003, p. 547.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, *Myrinet: A Gigabit-per-Second Local Area Network*, IEEE Micro **15** (1995), no. 1, 29–36.

[Bel97] Frank Bellosa, *Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment*, Technical Report TR-I4-02-97, IMMD IV - Department of Computer Science, University of Erlangen-Nürnberg, Jul 1997.

[BICrT08] Javier Garcia Blas, Florin Isaila, Jesus Carretero, and rossemann Thomas, *Implementation and Evaluation of an MPI-IO Interface for GPFS in ROMIO*, Proceedings of the 15th EuroPVM/MPI 2008 Conference (Dublin, Ireland), 2008.

[BJM+96] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes, *An implementation of the hamlyn sender-managed interface archi-*

tecture, OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation (New York, NY, USA), ACM, 1996, pp. 245–259.

[BRB98a] Raoul Bhoedjang, Tim Rühl, and Henri E. Bal, *Efficient Multicast on Myrinet using Link-Level Flow Control*, ICPP '98: Proceedings of the 1998 International Conference on Parallel Processing (Washington, DC, USA), IEEE Computer Society, 1998, p. 381.

[BRB98b] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal, *User-Level Network Interface Protocols*, Computer **31** (1998), 53–60.

[CCkL+02] Avery Ching, Alok Choudhary, Wei keng Liao, Rob Ross, and William Gropp, *Noncontiguous I/O through PVFS*, Proceedings of the IEEE International Conference on Cluster Computing (Los Alamitos, CA, USA), IEEE Computer Society, 2002, p. 405.

[CKE+05] Gyu Sang Choi, Jin-Ha Kim, D. Ersoz, M.S. Yousif, and C.R. Das, *Exploiting NIC Memory for Improving Cluster-Based Webserver Performance*, Proceedings of the IEEE International Conference on Cluster Computing, September 2005, pp. 1–10.

[CMC98] Brent N. Chun, Alan M. Mainwaring, and David E. Culler, *Virtual Network Transport Protocols for Myrinet*, IEEE Micro **18** (1998), no. 1, 53–63.

[CRU03] Olivier Cozette, Cyril Randriamaro, and Gil Utard, *READ^2: Put Disks at Network Level*, CCGRID'03, Workshop on Parallel I/O (Tokyo (Japan)), May 2003.

[Cyc] Cyclone, *PCI-X 740 Intelligent Dual Gigabit Ethernet Controller Datasheet*, http://www.cyclone.com/pdf/PCIX740.pdf.

[DBC+97] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li, *VMMC-2: Efficient Support for Reliable, Connection-oriented Communication*, Proceedings of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V) (Stanford, CA, USA), 1997.

[DRM+98] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd, *The virtual interface architecture*, IEEE Micro **18** (1998), no. 2, 66–76.

[Dyd01] Mads Bondo Dydensborg, *Direct Remote Access to Devices*, Proceedings of the Fourth European Research Seminar on Advanced Distributed Systems, May 2001.

[Ell07] Lars Ellenberg, *DRBD 8.0.x and beyond: Shared-Disk Semantics on a Shared-Nothing Cluster*, LinuxConf Europe 2007 (Cambridge, England), September 2007.

[Fas06] Mark Fasheh, *OCFS2: The Oracle Clustered File System, Version 2*, Proceedings of the 2006 Linux Symposium, July 2006, pp. 289–302.

[FB05] Michail D. Flouris and Angelos Bilas, *Violin: A Framework for Extensible Block-Level Storage*, MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (Washington, DC, USA), IEEE Computer Society, 2005, pp. 128–142.

[FLB08] M.D. Flouris, R. Lachaize, and A. Bilas, *Orchestra: Extensible Block-Level Support for Resource and Data Sharing in Networked Storage Systems*, ICPADS '08: Proceedings of the 14th International Conference on Parallel and Distributed Systems, December 2008, pp. 237–244.

[GAB+] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnsen, H. Kohmann, R. Nordstrom, and P. Werner, *Low Level SCI software functional specification-Software Infrastructure for SCI*, ESPRIT Project 23174, http://www.dolphinics.com/downloads/nt/pdf_zip/SISCI_API-2_1_1.pdf.

[GCM+08] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, , and Anna Toncheva, *The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011*, Idc white paper, sponsored by emc, International Data Corporation, Mar 2008.

[Geo02] Patrick Geoffray, *OPIOM: Off-Processor I/O with Myrinet*, Future Gener. Comput. Syst. **18** (2002), no. 4, 491–499.

[GKA+09] Georgios I. Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris, *Performance Evaluation of the Sparse Matrix-Vector Multiplication on Modern Architectures*, The Journal of Supercomputing **50** (2009), no. 1, 36–77.

[GNA+98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka, *A Cost-effective, High-bandwidth Storage Architecture*, ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ACM, 1998, pp. 92–103.

[Gor04] Mel Gorman, *Understanding the Linux Virtual Memory Manager*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[GP04] Brice Goglin and Loïc Prylli, *Performance Analysis of Remote File System Access over a High-Speed Local Network*, Proceedings of the Workshop on Communication Architecture for Clusters (CAC'04), held in conjunction with the 18th IEEE IPDPS Conference (Santa Fe, New Mexico), IEEE Computer Society Press, April 2004, p. 185.

[GPG04] Brice Goglin, Loïc Prylli, and Olivier Glück, *Optimizations of Client's side communications in a Distributed File System within a Myrinet Cluster*, Proceedings of the IEEE Workshop on High-Speed Local Networks (HSLN), held in conjunction with the 29th IEEE LCN Conference (Tampa, Florida), IEEE Computer Society Press, November 2004, pp. 726–733.

[Gur09] Sudhanva Gurumurthi, *Architecting Storage for the Cloud Computing Era*, IEEE Micro **29** (2009), 68–71.

[Han01] Jorgen S. Hansen, *Flexible Network Attached Storage using RDMA*, Proceedings of the 9th IEEE Symposium on High-Performance Interconnects, 2001.

[Hel99] H. Hellwagner, *The SCI Standard and Applications of SCI*, Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters (H. Hellwagner and A. Reinefield, eds.), Springer-Verlag, Sep 1999, pp. 3–34.

[HH05] Dean Hildebrand and Peter Honeyman, *Exporting Storage Systems in a Scalable Manner with pNFS*, MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (Washington, DC, USA), IEEE Computer Society, 2005, pp. 18–27.

[HL02] Jorgen S. Hansen and Renaud Lachaise, *Using Idle Disks in a Cluster as a High-Performance Storage System*, Proceedings of the IEEE International Conference on Cluster Computing, 2002.

[ID01] Sitaram Iyer and Peter Druschel, *Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O*, SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (New York, NY, USA), ACM, 2001, pp. 117–130.

[Inf00] InfiniBand Trade Association, *InfiniBand Architecture Specification, Release 1.0*, 2000, http://www.infinibandta.org/specs.

[Int] Intel Corporation, *Intel 80331 I/O Processor Datasheet*, http://www.intel.com/design/iio/datashts/273943.htm.

[Jan01] Jeff Janzen, *Calculating Memory System Power for DDR SDRAM*, Designline **10** (2001), no. 2.

[KK05] Evangelos Koukis and Nectarios Koziris, *Memory Bandwidth Aware Scheduling for SMP Cluster Nodes*, Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), 2005, pp. 187–196.

[KK06] Evangelos Koukis and Nectarios Koziris, *Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs*, ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems (Washington, DC, USA), IEEE Computer Society, 2006, pp. 345–354.

[KKJ02] Kangho Kim, Jin-Soo Kim, and Sung-In Jung, *GNBD/VIA: A Network Block Device over Virtual Interface Architecture on Linux.*, Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS), 2002.

[KLS86] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker, *VAXcluster: A Closely-coupled Distributed System*, ACM Transactions on Computer Systems (TOCS) **4** (1986), no. 2, 130–146.

[KPR02] Hyong-youb Kim, Vijay S. Pai, and Scott Rixner, *Increasing Web Server Throughput with Network Interface Data Caching*, Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X) (New York, NY, USA), ACM, October 2002, pp. 239–250.

[LPB04] Jiuxing Liu, Dhabaleswar K. Panda, and Mohammad Banikazemi, *Evaluating the Impact of RDMA on Storage I/O over Infiniband*, SAN-03 Workshop (in conjunction with HPCA), 2004.

[LPSS03] James Lentini, Vu Pham, Steven Sears, and Randall Smith, *Implementation and Analysis of the User Direct Access Programming Library*, 2nd Workshop on Novel Uses of System Area Networks, 2003.

[LSC+01] Tirthankar Lahiri, Vinay Srihari, Wilson Chan, N. MacNaughton, and Sashikanth Chandrasekaran, *Cache Fusion: Extending Shared-Disk Clusters with Shared Caches*, VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 2001, pp. 683–686.

[LT96] Edward K. Lee and Chandramohan A. Thekkath, *Petal: Distributed Virtual Disks*, ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA), ACM, 1996, pp. 84–92.

[LVE00] Jochen Liedtke, Marcus Völp, and Kevin Elphinstone, *Preliminary Thoughts on Memory-Bus Scheduling*, EW 9: Proceedings of the 9th

workshop on ACM SIGOPS European workshop, ACM Press, 2000, pp. 207–210.

[LYP06] Shuang Liang, Weikuan Yu, and Dhabaleswar K. Panda, *High Performance Block I/O for Global File System (GFS) with Infiniband RDMA*, ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing (Washington, DC, USA), IEEE Computer Society, 2006, pp. 391–398.

[MACM05] Daniel Mossé, Nevine AbouGhazaleh, Bruce R. Childers, and Rami G. Melhem, *Energy Conservation in Memory Hierarchies using Power-Aware Cached-DRAM*, Power-aware Computing Systems, 2005.

[Mar] Marvell Technology Group Ltd, *88F6281 Integrated Controller Hardware Specifications*, http://www.marvell.com/products/processors/embedded/kirkwood/HW_-88F6281_OpenSource.pdf.

[MPB07] Manolis Marazakis, Vassilis Papaefstathiou, and Angelos Bilas, *Optimization and Bottleneck Analysis of Network Block I/O in Commodity Storage Systems*, ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing (New York, NY, USA), ACM, 2007, pp. 33–42.

[MXPB06] Manolis Marazakis, Konstantinos Xinidis, Vassilis Papaefstathiou, and Angelos Bilas, *Efficient Remote Block-level I/O over an RDMA-capable NIC*, ICS '06: Proceedings of the 20th annual international conference on Supercomputing (New York, NY, USA), ACM, 2006, pp. 97–106.

[Myr03] Myricom, *GM: A Message-Passing System for Myrinet Networks*, 2003, http://www.myri.com/scs/GM-2/doc/html/.

[NC] William D. Norcott and Don Capps, *IOzone Filesystem Benchmark*, http://www.iozone.org/docs/IOzone_msword_98.pdf.

[NSM04] David Nagle, Denis Serenyi, and Abbie Matthews, *The Panasas Activescale Storage Cluster: Delivering Scalable High Bandwidth Storage*, SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2004, p. 53.

[osd04] *Object-Based Storage Device Commands (OSD)*, 2004, ANSI standard INCITS 400-2004.

[PBB+99] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe, *A 64-bit, Shared Disk File System for Linux*, Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems (San Diego, CA), 1999, pp. 22–41.

[PcFH+01] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg, *Quadrics Network (QsNet): High-Performance Clustering Technology*, Hot Interconnects 9 (Stanford University, Palo Alto, CA), August 2001.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, *IO-Lite: A Unified I/O Buffering and Caching System*, ACM Transactions on Computer Systems (TOCS) **18** (2000), no. 1, 37–66.

[Pet04] Mikael Pettersson, *The Perfctr Linux Performance Monitoring Counters Driver*, 2004, http://user.it.uu.se/~mikpe/linux/perfctr/.

[PFB09] George Panagiotakis, Michail D. Flouris, and Angelos Bilas, *Reducing Disk I/O Performance Sensitivity for Large Numbers of Sequential Streams*, ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (Washington, DC, USA), IEEE Computer Society, 2009, pp. 22–31.

[PLC95] Scott Pakin, Mario Lauria, and Andrew Chien, *High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet*, Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) (New York, NY, USA), ACM, 1995, p. 55.1.

[PSC03] Gang Peng, Srikant Sharma, and Tzi-cker Chiueh, *A Case for Network-Centric Buffer Cache Organization*, Procedings of the 11th Symposium on High Performance Interconnects, August 2003, pp. 66–71.

[PSC05] Gang Peng, Srikant Sharma, and Tzi-cker Chiueh, *Network-Centric Buffer Cache Organization*, ICDCS '05: Proceedings of the 25th IEEE Inter-

national Conference on Distributed Computing Systems (ICDCS'05) (Washington, DC, USA), IEEE Computer Society, 2005, pp. 219–228.

[PT97] L. Prylli and B. Tourancheau, *Protocol Design for High Performance Networking: A Myrinet Experience*, Technical Report 97-22, LIP-ENS Lyon, 69364 Lyon, France, 1997.

[PTH+01] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges, *MPI-IO/GPFS, An Optimized Implementation of MPI-IO on top of GPFS*, Proceedings of the 2001 ACM/IEEE conference on Supercomputing (New York, NY, USA), ACM, 2001, pp. 17–17.

[Ros] Robert Ross, *Parallel I/O Benchmarking Consortium*, http://www.mcs.anl.gov/research/projects/pio-benchmark.

[Sch03] Sebastian Schönberg, *Impact of PCI-Bus Load on Applications in a PC Architecture*, RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium (Washington, DC, USA), IEEE Computer Society, 2003, p. 430.

[SDH+96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck, *Scalability in the XFS File System*, ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference (Berkeley, CA, USA), USENIX Association, 1996, pp. 1–1.

[SE04] Laura Shepard and Eric Eppe, *SGI InfiniteStorage Shared Filesystem CXFS: A High-Performance, Multi-OS Filesystem*, Tech. report, Silicon Graphics Inc., 2004, http://www.sgi.com/pdfs/2691.pdf.

[SH98] I. Schoinas and M. Hill, *Address Translation Mechanisms in Network Interfaces*, HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture (Washington, DC, USA), IEEE Computer Society, 1998, p. 219.

[SH02] Frank Schmuck and Roger Haskin, *GPFS: A Shared-Disk File System for Large Computing Clusters*, Proc. of the First Conference on File and Storage Technologies (FAST), January 2002, pp. 231–244.

[SRO96] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe, *The Global File System*, Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems (College Park, MD), IEEE Computer Society Press, 1996, pp. 319–342.

[Sun08] Sun Microsystems, *Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System*, 2008, https://www.sun.com/offers/details/LustreFileSystem.xml.

[Sym] Symantec, *Veritas Storage Foundation Cluster Filesystem*, http://www.symantec.com/business/storage-foundation-cluster-file-system.

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, *Frangipani: A Scalable Distributed File System*, SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM, 1997, pp. 224–237.

[TOHI98] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa, *Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication*, IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium (Washington, DC, USA), IEEE Computer Society, 1998, p. 308.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels, *U-Net: A User-level Network Interface for Parallel and Distributed Computing*, SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles (New York, NY, USA), ACM, 1995, pp. 40–53.

[vEV98] Thorsten von Eicken and Werner Vogels, *Evolution of the Virtual Interface Architecture*, Computer **31** (1998), no. 11, 61–68.

[WS06] Jonathan Weinberg and Allan Snavely, *User-guided Symbiotic Space-sharing of Real workloads*, ICS '06: Proceedings of the 20th annual international conference on Supercomputing (New York, NY, USA), ACM, 2006, pp. 345–352.

[WUA+08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou, *Scalable Performance of the Panasas Parallel File System*, FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies (Berkeley, CA, USA), USENIX Association, 2008, pp. 1–17.

[WWP03] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda, *Supporting Efficient Noncontiguous Access in PVFS over InfiniBand*, Proceedings of the IEEE International Conference on Cluster Computing (Los Alamitos, CA, USA), IEEE Computer Society, 2003, p. 344.

[WWPR04] Jiesheng Wu, Pete Wyckoff, Dhabaleswar Panda, and Rob Ross, *Unifier: Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand*, Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04), IEEE Press, 2004.

[WYMG09] Jun Wang, Xiaoyu Yao, Christopher Mitchell, and Peng Gu, *A New Hierarchical Data Cache Architecture for iSCSI Storage Server*, IEEE Transactions on Computers **58** (2009), 433–447.

[yKRP05] Hyong youb Kim, Scott Rixner, and Vijay S. Pai, *Network Interface Data Caching*, IEEE Transactions on Computers **54** (2005), 1394–1408.

[YP05] W. Yu and D. K. Panda, *Benefits of Quadrics Scatter/Gather to PVFS2 Noncontiguous I/O*, International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI), September 2005.

Publications

- Evangelos Koukis, Anastassios Nanos, and Nectarios Koziris, *GMBlock: Optimizing Data Movement in a Block-level Storage Sharing System over Myrinet*, Journal of Cluster Computing, *under publication*.

- Evangelos Koukis, Anastassios Nanos, and Nectarios Koziris, *Synchronized Send Operations for Efficient Streaming Block I/O over Myrinet*, Proceedings of the Workshop on Communication Architecture for Clusters (CAC 2008), held in conjunction with the 22nd International Parallel and Distributed Processing Symposium (IPDPS 2008), 2008.

- Evangelos Koukis and Nectarios Koziris, *Efficient Block Device Sharing over Myrinet with Memory Bypass*, Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), 2007, p. 29.

- Evangelos Koukis and Nectarios Koziris, *Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps*, ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems (Washington, DC, USA), IEEE Computer Society, 2006, pp. 345–354.

- Evangelos Koukis and Nectarios Koziris, *Memory Bandwidth Aware Scheduling for SMP Cluster Nodes*, Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), 2005, pp. 187–196.

- Maria Athanasaki, Evangelos Koukis and Nectarios Koziris, *Scheduling of Tiled Nested Loops onto a Cluster with a Fixed Number of SMP Nodes*, Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP'04), 2004, pp. 424-433.

- Maria Athanasaki, Evangelos Koukis and Nectarios Koziris, *Efficient Scheduling of Tiled Iteration Spaces onto a Fixed Size Parallel Architecture*, Proceedings of the 9th Panhellenic Conference in Informatics, 2003, pp. 178-192.