Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τεχνικές caching για μνήμες τύπου Flash

Βαβλάς Ιωάννης

Επιβλέποντες καθηγητές

Κατσαρός Δημήτριος

Μποζάνης Παναγιώτης

ΒΟΛΟΣ
2010

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΛΗΨΗ

Οι περισσότερες φορητές συσκευές είναι εξοπλισμένες με μνήμη Flash τύπου NAND παρόλο που ανάμεσα στα χαρακτηριστικά της συγκαταλέγονται το not-in-place update και οι ασύμμετρες καθυστερήσεις Εισόδου/Εξόδου (I/O) μεταξύ των λειτουργιών ανάγνωσης, εγγραφής και διαγραφής : οι λειτουργίες εγγραφής και διαγραφής είναι πολύ πιο αργές και χρονοβόρες από εκείνη της ανάγνωσης στη μνήμη Flash. Για τη συνολική απόδοση ενός συστήματος που διαθέτει μνήμη Flash, η πολιτική αντικατάστασης buffer (buffer replacement policy) θα πρέπει να λάβει υπόψη τις παραπάνω ασύμμετρες καθυστερήσεις Εισόδου/Εξόδου (I/O) και το κόστος αντικατάστασης των βρόμικων σελίδων-θυμάτων. Οι υπάρχοντες αλγόριθμοι αντικατάστασης buffer (buffer replacement algorithms) όπως οι LRU, LIRS, ARC δεν μπορούν να αντιμετωπίσουν τα παραπάνω προβλήματα γιατί έχουν σχεδιαστεί για συστήματα αποθήκευσης με βάση το σκληρό δίσκο και ως κύριο μέλημα έχουν τη μεγιστοποίηση του ποσοστού επιτυχίας (hit ratio) στη μνήμη. Στο παρόν έγγραφο προτείνεται μια πολιτική αντικατάστασης buffer (buffer replacement policy) που ενισχύει τον αλγόριθμο αντικατάστασης LIRS με την αναδιάταξη των εγγραφών των ζεστών βρόμικων σελίδων (not-cold dirty pages) από την buffer cache προς τη μνήμη Flash. Ο ενισχυμένος αλγόριθμος LIRS-WSR επικεντρώνεται στην ελάττωση του πλήθους λειτουργιών εγγραφής/διαγραφής, ενώ ταυτόχρονα αποτρέπει τη μεγάλη μείωση του ποσοστού επιτυχίας στον buffer. Τα αποτελέσματα της οδηγούμενης από ίχνη προσομοίωσης (trace-driven simulation) δείχνουν ότι μεταξύ των υπαρχόντων αλγορίθμων αντικατάστασης buffer όπως των LRU, CF-LRU, ARC και LIRS, ο LIRS-WSR είναι ο βέλτιστος σχεδόν σε όλες τις περιπτώσεις για τα συστήματα με μνήμη Flash.

1. Εισαγωγή

Η μνήμη Flash είναι μνήμη τύπου EEPROM (electrically erasable and programmable read-only memory), η οποία έχει πολλά ελκυστικά χαρακτηριστικά, όπως χαμηλή κατανάλωση ενέργειας, αντοχή σε κραδασμούς, χαμηλό βάρος, μικρό μέγεθος, μεγάλη πυκνότητα, χαμηλό θόρυβο και υψηλή απόδοση Εισόδου/Εξόδου (I/O). Καθώς μειώνεται η τιμή της, ενώ παράλληλα αυξάνεται η χωρητικότητα της, υπάρχει μια ισχυρή τάση αντικατάστασης του μαγνητικού σκληρού δίσκου με μνήμη Flash για το δευτερεύων σύστημα αποθήκευσης (secondary storage) σε φορητές κυρίως υπολογιστικές συσκευές. Η μνήμη Flash χρησιμοποιείται ευρέως σε κινητά τηλέφωνα (smart phones), MP3/MP4 players, ψηφιακές φωτογραφικές μηχανές, PDAs (personal digital assistant), tablet PC's, φορητά DVD players, IP-phones, οικιακά gateways και φορητούς υπολογιστές. Πρόσφατα, η μνήμη Flash έχει υιοθετηθεί από προσωπικούς υπολογιστές και εξυπηρέτες (servers) ως on-board cache και Solid-State Disk (SSD).

Τα σύγχρονα λειτουργικά συστήματα υποστηρίζουν ένα μηχανισμό buffer για να ενισχύσουν την απόδοση του συστήματος, η οποία περιορίζεται από τις σχετικά αργές λειτουργίες του συστήματος δευτερεύουσας αποθήκευσης. Η buffer cache είναι πτητική μνήμη, δηλαδή δεν διατηρεί τα δεδομένα μετά την διακοπή της παροχής ρεύματος, και βρίσκεται μεταξύ του συστήματος αρχείων (file system) και της μνήμης Flash. Η buffer cache μπορεί να μειώσει τον αριθμό των αιτήσεων εγγραφής στη μνήμη Flash, συγχωνεύοντας επαναλαμβανόμενες αιτήσεις εγγραφής στην ίδια σελίδα. Ακόμα και για τα ενσωματωμένα λειτουργικά συστήματα, τα οποία δεν υποστηρίζουν μηχανισμό buffer, ένα ιδιόκτητο σχήμα buffering χρησιμοποιείται από τη συσκευή. Εξαιτίας των αργών αναγνώσεων από το σύστημα δευτερεύουσας αποθήκευσης, το λειτουργικό σύστημα αντιγράφει τα δεδομένα στον buffer και εξυπηρετεί τις επόμενες λειτουργίες ανάγνωσης από την ταχύτερη κύρια μνήμη. Για παράδειγμα, ένα MP3 Player διαθέτει ένα DRAM buffer μεγέθους 32 MBytes και αρκετά GBytes μνήμης Flash για την αποθήκευση των αρχείων multimedia. Διαβάζει ή εγγράφει πολλά αρχεία στον buffer, καθώς οι προσβάσεις στη μνήμη DRAM είναι αρκετά πιο γρήγορες από τη μνήμη Flash, ειδικά για τις λειτουργίες εγγραφής. Στην περίπτωση που το αποθηκευτικό μέσο είναι ένας σκληρός δίσκος, ελαχιστοποιείται επίσης η κατανάλωση ενέργειας, απενεργοποιώντας το σκληρό δίσκο. Για τις ψηφιακές φωτογραφικές μηχανές, η αποθήκευση μιας εικόνας στη μνήμη RAM πριν την εγγραφή της στη μνήμη Flash, μειώνει σημαντικά τον χρόνο αντίδρασης (response time), ο οποίος είναι από τους πιο σημαντικούς παράγοντες της απόδοσης για τις συγκεκριμένες συσκευές.

Οι πρόσφατες εφαρμογές για τη μνήμη Flash ποικίλουν, ενώ παράλληλα είναι αρκετά πολύπλοκες. Για παράδειγμα, τα κινητά τηλέφωνα τελευταίας γενιάς προσφέρουν στο χρήστη πλήρεις υπηρεσίες ηλεκτρονικού ταχυδρομείου (e-mail) και περιήγησης (browsing) στο Web. Για να γίνει αυτό, οι εφαρμογές πρέπει να αποθηκεύουν πολλά προσωρινά αρχεία του internet ή μικρά αρχεία e-mail στη μνήμη Flash. Επιπλέον, οι εφαρμογές αυτές μπορούν να εκτελούνται ταυτόχρονα, παράγοντας έτσι μικτές αιτήσεις εγγραφής.

Η μνήμη Flash παρουσιάζει πολύ καλύτερη απόδοση για τυχαίες αιτήσεις ανάγνωσης, συγκριτικά με τους μαγνητικούς σκληρούς δίσκους, διότι δεν διαθέτει μηχανικά μέρη, οπότε δεν έχει καθυστέρηση αναζήτησης (seek time). Σε ένα σκληρό

δίσκο, η καθυστέρηση αυτή μπορεί να διαρκέσει αρκετά milliseconds. Για διαδοχικές αιτήσεις ανάγνωσης και εγγραφής, η μνήμη Flash έχει παρόμοια ή ακόμα και καλύτερη απόδοση από ένα σκληρό δίσκο. Ωστόσο, για τυχαίες αιτήσεις εγγραφής η μνήμη Flash εμφανίζει κατώτερες επιδόσεις σε σχέση με τους σκληρούς δίσκους, καθώς διαθέτει αρκετούς περιορισμούς σε επίπεδο hardware. Πιο συγκεκριμένα, η μονάδα δεδομένων (data unit) της λειτουργίας διαγραφής για τη μνήμη Flash είναι το block, το οποίο αποτελείται από έναν προκαθορισμένο αριθμό συνεχόμενων σελίδων, παρόλο που η μονάδα δεδομένων των λειτουργιών ανάγνωσης/εγγραφής είναι η σελίδα. Επιπλέον, είναι αδύνατο να επανεγγράψουμε μια σελίδα στην ίδια θέση (in-place) στη μνήμη Flash. Επομένως, για να ενημερώσουμε (update) τα δεδομένα μιας σελίδας, το σύστημα θα πρέπει να πραγματοποιήσει μόνο ένα από τα παρακάτω :
1). Να εγγράψει τα δεδομένα σε μια νέα allocated σελίδα και να καταργήσει την αυθεντική σελίδα.
2). Να εγγράψει τα δεδομένα στην αυθεντική σελίδα, αφού πρώτα διαγράψει το block που περιέχει τη συγκεκριμένη σελίδα.
Στην τελευταία περίπτωση, είναι δύσκολο να διατηρηθεί η συνοχή των δεδομένων (data consistency). Στην πρώτη περίπτωση, η επαναχρησιμοποίηση άκυρων σελίδων για ανάγνωση/διαγραφή απαιτεί τη διαγραφή των blocks που περιέχουν τις συγκεκριμένες σελίδες.
Ακόμη, ο χρόνος ζωής της μνήμης Flash είναι μικρότερος από εκείνον ενός σκληρού δίσκου ή μιας μνήμης DRAM. Με άλλα λόγια, μόνο ένας περιορισμένος αριθμός λειτουργιών διαγραφής μπορούν να εκτελεστούν με ασφάλεια σε κάθε κελί μνήμης, τυπικά μεταξύ 100.000 και 1.000.000 κύκλων. Τέλος, υπάρχουν διαφορές μεταξύ των καθυστερήσεων Εισόδου/Εξόδου ανάλογα με τον τύπο της λειτουργίας, δηλαδή ανάγνωσης, εγγραφής και διαγραφής. Η λειτουργία εγγραφής είναι περίπου 10 φορές πιο αργή από τη λειτουργία ανάγνωσης, ενώ η λειτουργία διαγραφής είναι περίπου 20 φορές πιο αργή από τη λειτουργία εγγραφής.

Η τεχνική του Disk caching έχει χρησιμοποιηθεί για τη μείωση της καθυστέρησης Εισόδου/Εξόδου του δίσκου. Ένας αλγόριθμος αντικατάστασης buffer για ένα δίσκο, προσπαθεί να διατηρήσει τη βέλτιστη ακολουθία Εισόδου/Εξόδου από την αρχική ακολουθία, μειώνοντας το πλήθος των προσβάσεων (accesses). Υπάρχει μεγάλος αριθμός αλγορίθμων αντικατάστασης buffer για δίσκους, όπως για παράδειγμα ο LRU, ο LIRS και ο ARC. Με χρήση ιχνών Εισόδου/Εξόδου, ο LIRS επιδεικνύει πολύ καλή απόδοση, αφού χρησιμοποιεί το IR (Inter-reference Recency) για την αναγνώριση των ζεστών/κρύων σελίδων. Επομένως, ο LIRS επιλέγεται καταρχάς ως ο αλγόριθμος βάσης, ο οποίος πρέπει να ενισχυθεί για τη μνήμη Flash.

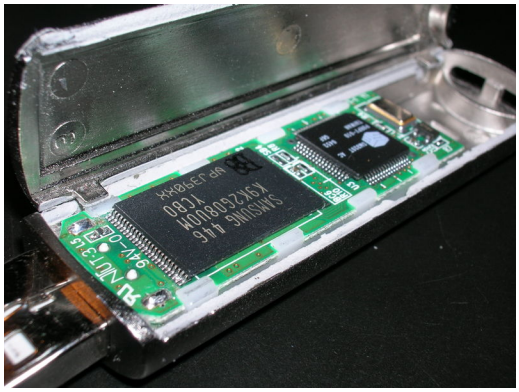
Αφού η μνήμη Flash αντικαθιστά τον σκληρό δίσκο, η τεχνική του Flash caching είναι αναγκαία για την ελάττωση των καθυστερήσεων Εισόδου/Εξόδου. Παρεμπιπτόντως, ένας αλγόριθμος αντικατάστασης buffer για μνήμη Flash πρέπει επιπροσθέτως να αντιμετωπίσει το πρόβλημα των διαφορετικών καθυστερήσεων Εισόδου/Εξόδου σύμφωνα με τον τύπο της λειτουργίας, δηλαδή ανάγνωσης, εγγραφής και διαγραφής, μολονότι είναι παρόμοιος με τους αλγόριθμους αντικατάστασης buffer για δίσκο. Επιχειρεί να διατηρήσει τη βέλτιστη ακολουθία Εισόδου/Εξόδου από την αρχική ακολουθία, μειώνοντας επιλεκτικά το πλήθος των προσβάσεων σύμφωνα με τον τύπο της λειτουργίας Εισόδου/Εξόδου. Αφού ο LIRS δεν λαμβάνει υπόψη του τις ασύμμετρες καθυστερήσεις Εισόδου/Εξόδου, επιδεικνύει φτωχότερη απόδοση σε σύστημα με μνήμη Flash σε σχέση με σύστημα με σκληρό δίσκο.

Επιπρόσθετα, αφού μια λειτουργία διαγραφής δεν ελέγχεται απευθείας από το buffer management layer αλλά από το κατώτερο στρώμα (layer), μια ακολουθία Εισόδου/Εξόδου παραγόμενη από έναν αλγόριθμο αντικατάστασης buffer για μνήμη Flash, αποτελείται μόνο από λειτουργίες ανάγνωσης/εγγραφής (απουσιάζουν οι λειτουργίες διαγραφής). Ευτυχώς, ο αριθμός των λειτουργιών εγγραφής από το buffer management layer είναι ανάλογος του αριθμού των φυσικών εγγραφών και διαγραφών στη μνήμη Flash. Επομένως, εστιάζουμε στην εύρεση ενός αλγορίθμου που να ελαχιστοποιεί το πλήθος των αιτήσεων για εγγραφή (write requests) αλλά και τις απώλειες του ποσοστού επιτυχίας (hit ratio), παράγοντας τη βέλτιστη ακολουθία Εισόδου/Εξόδου από την αρχική ακολουθία.

Ο LIRS-WSR ενισχύει τον αλγόριθμο αντικατάστασης buffer με μια πρόσθετη στρατηγική αντικατάστασης, η οποία ονομάζεται Write Sequence Reordering (WSR). Το κόστος αντικατάστασης προκύπτει όταν μια βρόμικη σελίδα, η οποία έχει τροποποιηθεί πριν την απομάκρυνση της από τον buffer, εγγράφεται στη μνήμη Flash για να ελευθερωθεί χώρος στον buffer. Οι λειτουργίες εγγραφής στη μνήμη Flash απαιτούν περισσότερο χρόνο και ενέργεια από τις λειτουργίες ανάγνωσης, ενώ μια αύξηση του αριθμού των λειτουργιών εγγραφής συνοδεύεται από ακόμα περισσότερο δαπανηρές λειτουργίες διαγραφής. Ο WSR επαναδιατάσει τις εγγραφές των ζεστών βρόμικων σελίδων από την buffer cache προς το σκληρό δίσκο, με σκοπό να μειώσει τον αριθμό των λειτουργιών εγγραφής, χωρίς ταυτόχρονα να υπάρξει μεγάλη υποβάθμιση του ποσοστού επιτυχίας. Μια υποβάθμιση του ποσοστού επιτυχίας στην buffer cache, έχει ως αποτέλεσμα ένα μεγάλο αριθμό αναγνώσεων από τη μνήμη Flash. Για την απρόσκοπτη ολοκλήρωση των LIRS και WSR, έχουν τροποποιηθεί όλα τα βήματα του LIRS, ενώ παράλληλα έχουν διατηρηθεί όλα τα πλεονεκτήματα του, όπως για παράδειγμα το IR. Ο συγκεκριμένος αλγόριθμος σχεδιάστηκε επίσης για να ελαχιστοποιήσει τόσο τα χρονικά (temporal), όσο τα χωρικά (spatial) κόστη που απαιτούνται για να επιτευχθεί ο στόχος. Η προσομοίωση δείχνει ότι ο LIRS-WSR μειώνει αποτελεσματικά τον αριθμό των φυσικών εγγραφών και διαγραφών, και κατά συνέπεια υπερισχύει άλλων αλγορίθμων.

2. Μνήμη Flash

Η μνήμη Flash (Εικόνα 2.1) είναι τύπου EEPROM (electrically erasable and programmable read-only memory) και είναι μη-πτητική, δηλαδή διατηρεί τα δεδομένα μετά την διακοπή της παροχής ρεύματος.



Εικόνα 2.1 : Ένα USB Flash Disk. Το chip στα αριστερά είναι η μνήμη Flash, ενώ δεξιά βρίσκεται ο μικρο-ελεγκτής.

Υπάρχουν δύο τύποι μνήμης Flash : η μνήμη Flash NOR και η μνήμη Flash NAND. Η μνήμη Flash NOR αναπτύχθηκε για να αντικαταστήσει τη μνήμη PROM (programmable read-only memory) και τη μνήμη EPROM (erasable PROM), οι οποίες χρησιμοποιούνταν για την αποθήκευση κώδικα (code storage). Έχει σχεδιαστεί για αποτελεσματικές τυχαίες προσπελάσεις και γρήγορες λειτουργίες ανάγνωσης, ενώ διαθέτει ξεχωριστούς διαύλους διευθύνσεων και δεδομένων, όπως η μνήμη EPROM και η SRAM (static random access memory). Εξαιτίας των παραπάνω χαρακτηριστικών της, η μνήμη Flash NOR χρησιμοποιείται συνήθως για την αποθήκευση του BIOS ενός υπολογιστή. Η μνήμη Flash NAND αναπτύχθηκε πιο πρόσφατα για την αποθήκευση δεδομένων, οπότε σχεδιάστηκε για να έχει πιο πυκνή αρχιτεκτονική και απλούστερη διεπαφή από τη μνήμη Flash NOR. Πιο συγκεκριμένα, η μνήμη Flash NAND διαθέτει μια διεπαφή Εισόδου/Εξόδου με εισόδους ελέγχου και χρησιμοποιείται ευρέως στις σημερινές υπολογιστικές συσκευές ως αντικατάστατο του σκληρού δίσκου, λόγω της μεγάλης χωρητικότητας, του χαμηλού κόστους και των ταχύτερων λειτουργιών εγγραφής της.

Τα χαρακτηριστικά της μνήμης Flash διαφέρουν σημαντικά από αυτά των μαγνητικών σκληρών δίσκων. Η μνήμη Flash δεν αποτελείται από μηχανικά μέρη, οπότε δεν παρουσιάζει καθυστέρηση που να προκαλείται από την κίνηση των μηχανικών κεφαλών με σκοπό να εντοπίσουν τη σωστή θέση για να διαβάσουν ή εγγράψουν δεδομένα. Στους μαγνητικούς σκληρούς δίσκους, η καθυστέρηση αυτή (seek time) είναι μία από τις πιο χρονοβόρες διαδικασίες στις δραστηριότητες Εισόδου/Εξόδου και τα λειτουργικά συστήματα έχουν πραγματοποιήσει διάφορες βελτιστοποιήσεις, όπως η προανάκτηση (prefetching) και ο χρονοπρογραμματισμός (scheduling) δίσκου με σκοπό να αποσβέσουν το κόστος.

Οι λειτουργίες ανάγνωσης και εγγραφής έχουν ασύμμετρα χαρακτηριστικά όσον αφορά την απόδοση και την κατανάλωση ενέργειας. Η καθυστέρηση ανάγνωσης της μνήμης Flash NOR είναι μικρότερη από εκείνη της NAND, αλλά οι καθυστερήσεις

εγγραφής και διαγραφής είναι πολύ μεγαλύτερες. Η αρχιτεκτονική NAND προσφέρει πολύ μεγάλη πυκνότητα κελιών, πράγμα που σημαίνει ότι η χωρητικότητα της μνήμης μπορεί να γίνει πολύ μεγάλη ενώ το μέγεθος της μνήμης να παραμείνει μικρό.

Πίνακας 1 : χαρακτηριστικά της μνήμης Flash

Device	current (mA)		Access time (4kB)		
	Idle	Active	Read	Write	Erase
NOR	0.03	32	20 us	28 ms	1.2 sec
NAND	0.01	10	25 us	250 us	2 ms

Η μνήμη Flash NAND υποστηρίζει Είσοδο/Έξοδο σε επίπεδο σελίδας, και η καθυστέρηση εγγραφής είναι περίπου 10 φορές μεγαλύτερη από την καθυστέρηση ανάγνωσης, όπως φαίνεται στον πίνακα 1. Οι λειτουργίες ανάγνωσης και εγγραφής πραγματοποιούνται σε σελίδες, οι οποίες έχουν μέγεθος συνήθως 512 Bytes. Οι λειτουργίες διαγραφής εφαρμόζονται σε blocks, τα οποία αποτελούνται συνήθως από 32 (ή 64) σελίδες το καθένα, οπότε έχουν μέγεθος 16 (ή 32) KBytes. Ωστόσο, τα παραπάνω μεγέθη δεν ισχύουν κατά κανόνα αλλά εξαρτώνται από τον τύπο της συσκευής αφού μια σελίδα στη μνήμη Flash NAND, η οποία είναι παρόμοια με έναν τομέα (sector) ενός σκληρού δίσκου, μπορεί να έχει μέγεθος 2 KBytes ή ακόμα και να φτάσει στα 4 KBytes, ενώ στη μνήμη Flash NOR, το μέγεθος μιας σελίδας είναι απλώς μια λέξη (word), δηλαδή 32 bits = 4 Bytes.

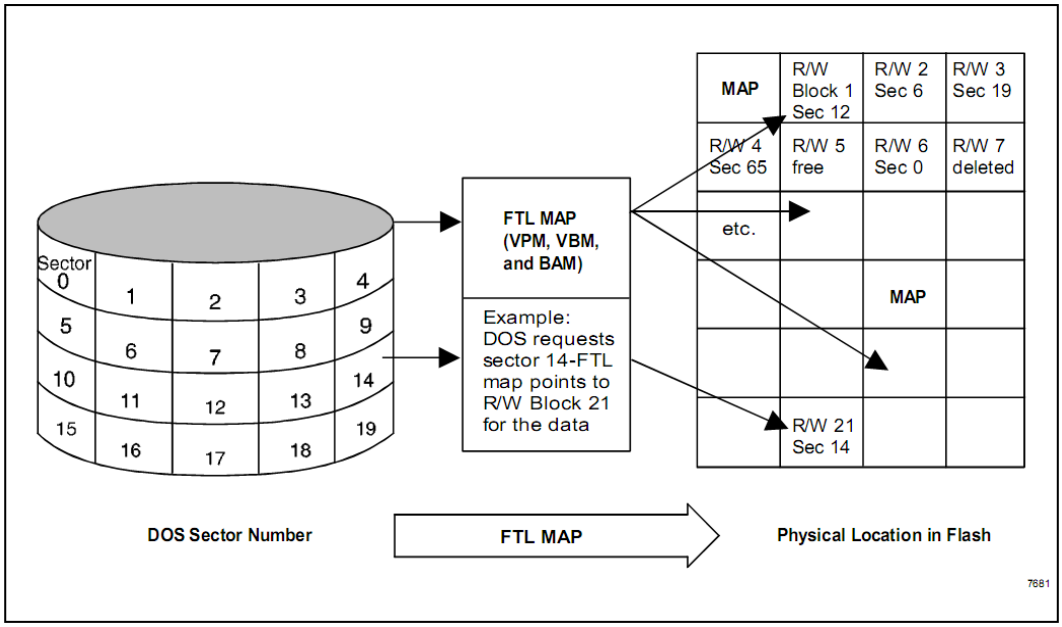
Οι μνήμες Flash έχουν ένα κοινό φυσικό περιορισμό. Πρέπει να διαγραφούν πριν εγγραφούν. Στη μνήμη Flash, η παρουσία ενός ηλεκτρικού φορτίου αναπαριστά το 1 ή το 0, και τα φορτία αυτά μπορούν να μεταφερθούν προς ή από ένα transistor, από μια λειτουργία διαγραφής ή εγγραφής. Γενικά, η λειτουργία διαγραφής, η οποία κάνει ένα κελί να αναπαριστά το 1, απαιτεί περισσότερο χρόνο από τη λειτουργία εγγραφής, οπότε η μονάδα λειτουργίας της σχεδιάστηκε να είναι μεγαλύτερη από εκείνη της λειτουργίας διαγραφής, για καλύτερες επιδόσεις. Άρα, η μνήμη Flash μπορεί να διαβαστεί ή να εγγραφεί κατά μία σελίδα τη φορά, ενώ διαγράφεται κάθε φορά κατά ένα block.

Επιπλέον, η μνήμη Flash πάσχει από έναν ακόμα περιορισμό, όσον αφορά τον αριθμό των λειτουργιών διαγραφής που μπορούν να πραγματοποιηθούν σε κάθε block. Το στρώμα μόνωσης (insulation layer), το οποίο αποτρέπει τη διασπορά των ηλεκτρικών φορτίων, μπορεί να καταστραφεί μετά από έναν αριθμό λειτουργιών διαγραφής. Στις single level cell (SLC) μνήμες Flash NAND, ο προσδοκώμενος αριθμός διαγραφών για κάθε block είναι 100.000, ενώ ο αντίστοιχος αριθμός μειώνεται στις 10.000 για τις multi level cell (MLC) μνήμες Flash NAND. Αν μερικά blocks που περιέχουν κρίσιμες πληροφορίες φθαρούν, αχρηστεύεται ολόκληρη η μνήμη παρόλο που υπάρχουν πολλά χρήσιμα blocks. Ως εκ τούτου, πολλές συσκευές που διαθέτουν μνήμη Flash χρησιμοποιούν τεχνικές wear-leveling για να διασφαλίσουν την ομοιόμορφη φθορά των blocks.

3. Flash Translation Layer (FTL)

3.1 Block-level mapping, Page-level mapping και Hybrid mapping FTL

Μια άλλη τεχνική που χρησιμοποιείται για τη βελτίωση της απόδοσης της μνήμης Flash, εκτός από την buffer cache, είναι το FTL (Flash Translation Layer). Αφού τα περισσότερα λειτουργικά συστήματα περιμένουν η δευτερεύουσα μνήμη να είναι ένας σκληρός δίσκος, δηλαδή μια συσκευή block, παρόλο που η μνήμη Flash δεν είναι τέτοια συσκευή, χρησιμοποιείται το FTL, το οποίο είναι ένα λεπτό στρώμα λογισμικού μεταξύ του λειτουργικού συστήματος και της μνήμης Flash. Το FTL μεταμφιέζει τη μνήμη Flash σε συσκευή block, ενώ αποκρύπτει παράλληλα, όσο μπορεί, τη διαφορά στην καθυστέρηση μεταξύ των λειτουργιών εγγραφής και διαγραφής της μνήμης Flash, καθώς δίνει την εντύπωση ότι τα δεδομένα εγγράφονται στην ίδια θέση, παρόλο που στην πραγματικότητα εγγράφονται σε άλλα σημεία της μνήμης. Η κύρια λειτουργία του FTL είναι η αντιστοίχιση της διεύθυνσης της λογικής σελίδας από το σύστημα αρχείων με τη διεύθυνση της φυσικής σελίδας στη μνήμη Flash. Επίσης, το FTL δημιουργεί μικρά εικονικά block δεδομένων, δηλαδή σελίδες ή τομείς (sectors), από τα μεγάλα block διαγραφής της μνήμης Flash, ενώ παράλληλα φροντίζει ώστε να υπάρχουν ελεύθερες/διαγραμμένες σελίδες στη μνήμη Flash για την αποθήκευση δεδομένων.



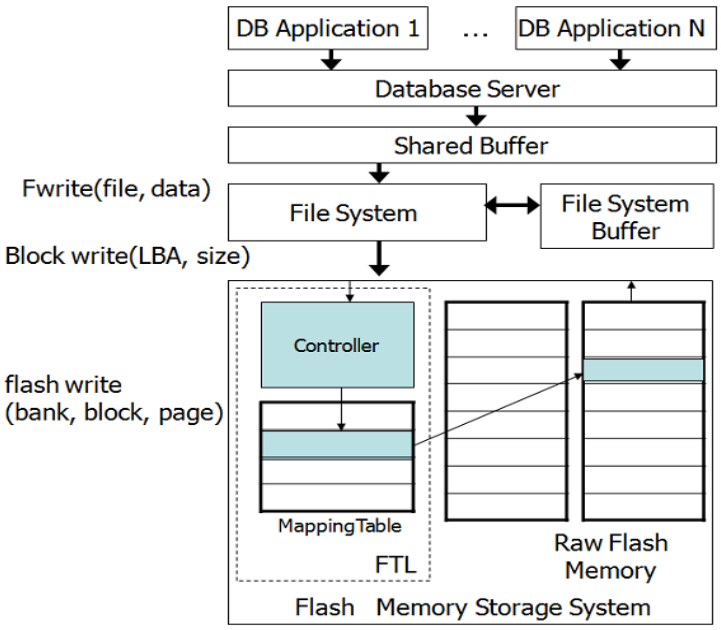
Εικόνα 3.1 : FTL Sector Relocation

Υπάρχουν διάφοροι τύποι αλγορίθμων FTL. Τα κινητά τηλέφωνα χρησιμοποιούν σχετικά πολύπλοκους αλγόριθμους, αλλά απλούστερες μέθοδοι εφαρμόζονται στους δίσκους USB Flash και τις κάρτες μνήμης Flash. Στην περίπτωση των Solid-State Disks, ο ελεγκτής διαθέτει περιορισμένη υπολογιστική δύναμη και μνήμη RAM για να διαχειριστεί μια μεγάλης χωρητικότητας μνήμη Flash, δεκάδων GigaByte.

Για να επιτύχουν καλύτερες επιδόσεις έχοντας στη διάθεση τους περιορισμένους πόρους, μερικά FTL εκμεταλλεύονται την τοπικότητα (locality) στις αιτήσεις εγγραφής. Ένα μικρό μέρος της μνήμης Flash χρησιμοποιείται ως buffer για τις αιτήσεις εγγραφών για να αντισταθμίσει τα μειονεκτήματα των φυσικών χαρακτηριστικών της μνήμης Flash. Ο μικρός αυτός buffer μπορεί να γίνει πολύ αποτελεσματικός για μεγάλη τοπικότητα στις αιτήσεις εγγραφών. Εν τούτοις, τα FTL παρουσιάζουν φτωχές επιδόσεις για τυχαίες εγγραφές χωρίς τοπικότητα. Οι χαμηλές επιδόσεις της μνήμης Flash για τυχαίες εγγραφές μπορεί επηρεάσουν σημαντικά τα συστήματα desktop ή server, τα οποία είναι πιθανό να έχουν πιο πολύπλοκα pattern εγγραφών, καθώς πολλαπλές διεργασίες Εισόδου/Εξόδου έχουν ως αποτέλεσμα πιο σύνθετα patterns εγγραφών.

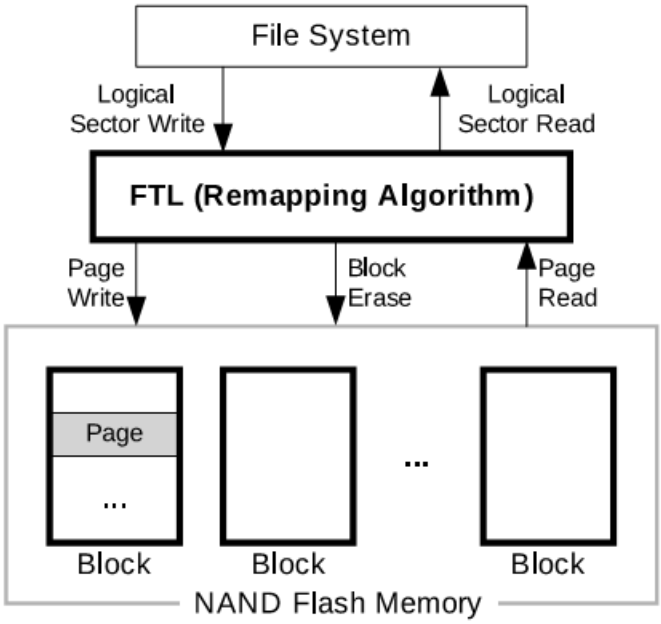
Πολλές διαφορετικές προσεγγίσεις υπάρχουν για τη βελτίωση των επιδόσεων των τυχαίων εγγραφών. Για παράδειγμα, πολλοί Solid-State Disks χρησιμοποιούν μια ξεχωριστή RAM μέσα στη συσκευή, η οποία λειτουργεί ως buffer. Το θέμα ωστόσο είναι η ορθή χρήση του RAM buffer. Στην περίπτωση του σκληρού δίσκου, ο αλγόριθμος του ανελκυστήρα χρησιμοποιείται για την ελαχιστοποίηση των κινήσεων των μηχανικών κεφαλών.

Το FTL αποθηκεύει στη μνήμη Flash ένα μέρος των δεδομένων αντιστοίχισης των λογικών σελίδων σε φυσικές σελίδες, μειώνοντας έτσι το κόστος της ενημέρωσης των παραπάνω αντιστοιχίσεων. Επιπλέον, αποθηκεύει τον πίνακα αντιστοίχισης στη μνήμη RAM με σκοπό τη γρήγορη μετάφραση των διευθύνσεων.



Εικόνα 3.2 : Αρχιτεκτονική συστήματος αποθήκευσης μνήμης Flash NAND

Όπως φαίνεται στην Εικόνα 3.2, το σύστημα αρχείων θεωρεί τη μνήμη Flash ως συσκευή block. Οι επανεγγραφές σελίδων μπορούν να πραγματοποιηθούν “λογικά” στο επίπεδο του συστήματος αρχείων. Παρ' όλα αυτά, επανεγγεγραμμένες σελίδες με την ίδια διεύθυνση εγγράφονται “φυσικά” σε διαφορετικές σελίδες ή ακόμα και σε διαφορετικά block. Επομένως, μειώνοντας το πλήθος των επανεγγραφών σελίδων στο επίπεδο του συστήματος αρχείων, μειώνεται και ο αριθμός των φυσικών εγγραφών και διαγραφών στη μνήμη Flash. Ως αποτέλεσμα, βελτιώνεται η απόδοση του συστήματος αρχείων και μεγαλώνει ο χρόνος ζωής της μνήμης Flash.



Εικόνα 3.3 : Το FTL εξομοιώνει τις λειτουργίες ανάγνωσης/εγγραφής τομέα ενός σκληρού δίσκου, για να μπορούν να χρησιμοποιηθούν συμβατικά συστήματα αρχείων (NTFS, FAT) στη μνήμη Flash NAND.

Τα σχήματα αντιστοίχισης διευθύνσεων του FTL μπορούν να διαιρεθούν σε τρεις κλάσεις : block-level mapping, page-level mapping και hybrid mapping.

Στο block-level mapping, ο πίνακας αντιστοίχισης διατηρεί τις πληροφορίες αντιστοίχισης μεταξύ της διεύθυνσης λογικού block και της διεύθυνσης φυσικού block. Οπότε, μια λογική σελίδα εγγράφεται με το σχήμα in-place, το οποίο σημαίνει ότι μια σελίδα εγγράφεται σε σταθερό σημείο ενός block, το οποίο καθορίζεται από τη μετατόπιση της σελίδας μέσα στο block. Το block-level mapping απαιτεί ένα μικρό σε μέγεθος πίνακα αντιστοίχισης. Όμως, ακόμα και όταν μια μικρή ποσότητα ενός block τροποποιηθεί, το συγκεκριμένο block πρέπει να διαγραφεί και οι μη ενημερωμένες όπως και οι ενημερωμένες σελίδες πρέπει να αντιγραφούν σε ένα νέο block. Ο περιορισμός αυτός έχει ως αποτέλεσμα ένα μεγάλο κόστος μετανάστευσης σελίδας, άρα χαμηλές επιδόσεις εγγραφών.

Στο page-level mapping, ο πίνακας αντιστοίχισης διατηρεί τις πληροφορίες αντιστοίχισης μεταξύ της διεύθυνσης λογικής σελίδας και της διεύθυνσης φυσικής σελίδας. Επομένως, μια λογική σελίδα μπορεί να αντιστοιχηθεί με το σχήμα out-of-place, το οποίο σημαίνει ότι μια λογική σελίδα μπορεί να εγγραφεί σε οποιαδήποτε

φυσική σελίδα ενός block. Αν σταλεί μια αίτηση ενημέρωσης για δεδομένα τα οποία έχουν ήδη εγγραφεί στη μνήμη Flash, το FTL εγγράφει τα δεδομένα σε μια διαφορετική, καθαρή σελίδα και στη συνέχεια τροποποιεί/ενημερώνει τις πληροφορίες αντιστοίχισης. Η παλιά σελίδα ακυρώνεται, μαρκάροντας το αντίστοιχο πεδίο της. Το μειονέκτημα του σχήματος αυτού είναι ότι το μέγεθος του πίνακα αντιστοίχισης είναι αναπόφευκτα μεγάλο. Το page-level mapping επαρκούσε για μικρού μεγέθους μνήμης Flash NAND, γιατί το μέγεθος του πίνακα αντιστοίχισης ήταν επίσης μικρό. Ωστόσο με την εκθετική αύξηση του μεγέθους της μνήμης Flash, η μέθοδος αυτή έγινε αναποτελεσματική. Σε μερικές περιπτώσεις, οι πληροφορίες αντιστοίχισης πρέπει να ανακατασκευαστούν, σαρώνοντας ολόκληρη τη μνήμη Flash κατά την εκκίνηση.

Το hybrid mapping χρησιμοποιεί από κοινού και το page mapping και το block mapping. Στο σχήμα αυτό, όλα τα φυσικά block της μνήμης Flash διαχωρίζονται σε log blocks και data blocks. Τα log blocks ονομάζονται και log buffer. Για αυτό το λόγο, το FTL που χρησιμοποιεί hybrid mapping λέγεται και log buffer-based FTL ή log-block FTL. Ενώ τα log blocks χρησιμοποιούν το page-level mapping και το σχήμα out-of-place, δηλαδή μια λογική σελίδα εγγράφεται σε οποιαδήποτε φυσική σελίδα ενός block, τα data blocks χρησιμοποιούν το block-level mapping και το σχήμα in-place, δηλαδή μια σελίδα εγγράφεται σε σταθερό σημείο ενός block. Ο αλγόριθμος του log-block FTL είναι ένας από τους πιο δημοφιλείς σήμερα, γιατί συνδυάζει την ανταγωνιστική απόδοση με το σχετικά χαμηλό κόστος, όσον αφορά τη χρήση της μνήμης RAM και την κατανάλωση ισχύος του επεξεργαστή.

3.2 Log-block FTL (hybrid mapping FTL ή log buffer-based FTL)

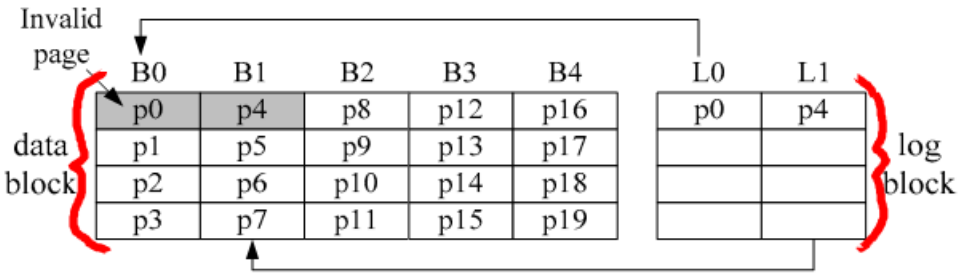
Για μια αίτηση εγγραφής, το log-block FTL στέλνει αρχικά τα δεδομένα σε ένα log block και ακυρώνει τα αντίστοιχα παλιά δεδομένα στο data block, όπου και είχαν αποθηκευτεί. Όταν τα log blocks γεμίσουν και δεν υπάρχει κενός χώρος, ένα log block επιλέγεται ως θύμα και όλες οι έγκυρες σελίδες στο log block μεταφέρονται σε data blocks, ούτως ώστε να δημιουργηθεί ελεύθερος χώρος για τις επόμενες αιτήσεις εγγραφής. Σε αυτό το βήμα, το log block συγχωνεύεται με τα data blocks που σχετίζεται. Έτσι λοιπόν, το βήμα αυτό ονομάζεται block merge ή garbage collection.

Υπάρχουν τρία είδη block merges : το full merge, το partial merge και το switch merge. Το partial merge και το switch merge μπορούν να πραγματοποιηθούν μόνο όταν όλες οι σελίδες στο log block θύμα εγγράφονται με το σχήμα in-place, δηλαδή όταν όλες οι σελίδες εγγράφονται σε καθορισμένη θέση. Για παράδειγμα, όταν όλες οι σελίδες εγγράφονται διαδοχικά από την πρώτη σελίδα του log block μέχρι την τελευταία, το log block αποκτά την ίδια κατάσταση με το data block. Σε αυτήν την περίπτωση, το log block μπορεί να αντικαταστήσει το data block και το παλιό data block μπορεί να ελευθερωθεί. Η αντικατάσταση αυτή λέγεται switch merge και είναι η ιδανική περίπτωση, όσον αφορά την απόδοση της μνήμης Flash, καθώς απαιτεί τη διαγραφή ενός μόνο block (παλιό data block) και την εγγραφή N σελίδων, όπου N είναι ο αριθμός των σελίδων για κάθε block. Σε αντίθετη περίπτωση,δηλαδή όταν για παράδειγμα η σελίδα 0 εγγράφεται επανειλημμένα 4 φορές και το κάθε block αποτελείται από 4 σελίδες, όλες οι σελίδες μπορούν να χρησιμοποιηθούν μόνο για τη σελίδα 0. Αφού λοιπόν γεμίσει το log block, οι έγκυρες σελίδες του παλιού data block και του log block (η σελίδα που περιέχει την τελευταία εγγραφή της σελίδας 0) αντιγράφονται σε ένα ελεύθερο data block, το οποίο με τη σειρά του γίνεται το νέο data block. Στη συνέχεια, το log block και το παλιό data block γίνονται ελεύθερα blocks. Η συγχώνευση αυτή είναι το full merge, το οποίο απαιτεί δύο διαγραφές block (log block και παλιό data block), N αναγνώσεις σελίδων και N εγγραφές σελίδων.

Για να βελτιωθεί η απόδοση κατά την εγγραφή της μνήμης Flash, το κόστος που προκαλείται από τις συγχωνεύσεις των block πρέπει να ελαττωθεί. Επομένως, τα πιο πρόσφατα log-block FTL στοχεύουν στη μείωση του αριθμού των block merges. Σχεδόν μέχρι και σήμερα, τα FTL επικεντρώνονται κυρίως σε pattern διαδοχικών εγγραφών, αφού στοχεύουν στα συστήματα πολυμέσων, όπως τα MP3 players και οι ψηφιακές φωτογραφικές μηχανές. Ωστόσο, στις πιο πρόσφατες συσκευές Flash, πολλές διεργασίες παράγουν διαδοχικές και τυχαίες αιτήσεις εγγραφών ταυτόχρονα. Αυτό έχει ως αποτέλεσμα πολλά FTL να έχουν φτωχές επιδόσεις, αφού οι τυχαίες εγγραφές προκαλούν συχνές συγχωνεύσεις των log blocks.

Υπάρχουν δύο είδη Log-block FTL : το FAST (Fully Associative Sector Translation) FTL και το BAST FTL, ανάλογα με την πολιτική συσχέτισης block. Η πολιτική συσχέτισης block έχει να κάνει με το πόσα data blocks μπορεί να σχετίζεται ένα log block. Στο σχήμα BAST (1 : 1), ένα log block σχετίζεται με ένα μόνο data block. Στην Εικόνα 5α, υπάρχουν 5 data blocks (B0, B1, B2, B3 και B4) και 2 log blocks (L0 και L1). Υποθέτουμε ότι κάθε block αποτελείται από 4 σελίδες. Όταν φτάσουν οι αιτήσεις ενημέρωσης για τις σελίδες p0 και p4, οι σελίδες εγγράφονται στα log blocks, ενώ ακυρώνονται οι σελίδες στα data blocks. Τα log blocks L1 και L2 σχετίζονται με τα

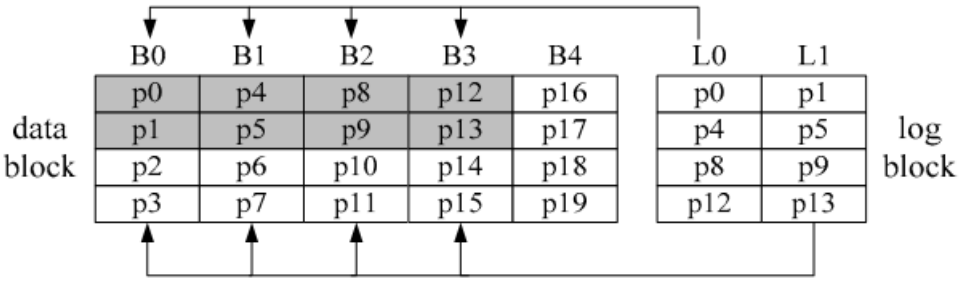
data blocks B0 και B1 αντίστοιχα. Οι σελίδες p0 και p4 εγγράφονται στα log blocks με τα οποία σχετίζονται αντίστοιχα.



Εικόνα 3.4α : 1:1 log block mapping (BAST)

Το σχήμα 1:1 (BAST) μπορεί να προκαλέσει συχνά log block merges. Για παράδειγμα, αν φτάσει η ακολουθία εγγραφών “p8, p12, p1, p5, p9, p13”, θα πρέπει να αντικατασταθεί ένα από τα log blocks παράγοντας μια δαπανηρή συγχώνευση για κάθε εγγραφή. Επιπλέον, κάθε log block θύμα, σε αυτό το παράδειγμα, περιέχει μία μόνο σελίδα όταν αντικαθίσταται, αφού οι υπόλοιπες σελίδες είναι κενές. Οπότε, τα log blocks στο σχήμα BAST παρουσιάζουν πολύ μικρή χρησιμοποίηση του χώρου όταν αντικαθιστώνται. Αν οι εγγραφές είναι τυχαίες, το σχήμα 1:1 επιδεικνύει φτωχή απόδοση αφού οι συχνές συγχωνεύσεις log block είναι αναπόφευκτες. Αυτό το φαινόμενο, όπου σχεδόν κάθε εγγραφή έχει ως αποτέλεσμα ένα block merge λέγεται λυγισμός (thrashing) των log blocks.

Για να αντιμετωπιστεί το πρόβλημα του λυγισμού των log blocks, προτάθηκε το σχήμα 1:N (FAST). Στο σχήμα αυτό, ένα log block μπορεί να σχετίζεται με πολλά data blocks όπως φαίνεται στην Εικόνα 5b.



Εικόνα 3.4b : 1:N log block mapping (FAST)

Χρησιμοποιώντας την αντιστοίχιση 1:N, για την ίδια ακολουθία εγγραφών “p8, p12, p1, p5, p9, p13”, δεν έχουμε block merge, ενώ στο σχήμα BAST κάθε εγγραφή προκαλεί block merge.

Ωστόσο, το πρόβλημα της σχήματος 1:N είναι η υψηλή συσχέτιση block, το οποίο σημαίνει ότι ότι ένα log block σχετίζεται με πολλά data blocks. Για παράδειγμα, όταν αντικαθίσταται το log block L1, χρειάζονται 16 αντίγραφα σελίδων (16 σελίδες = 4 blocks * 4 σελίδες) αφού το log block L1 σχετίζεται με 4 data blocks, B0, B1, B2 και

Β3. Αυτό σημαίνει ότι το σχήμα FAST απαιτεί μεγάλο κόστος για κάθε block merge, παρόλο που προκαλεί μικρό αριθμό συγχωνεύσεων.

Πρόσφατα, προτάθηκε ένα ειδικό σχήμα N:N, όπου N log blocks μπορούν να σχετίζονται με N data blocks και είναι ένα υβριδικό σχήμα όπου χρησιμοποιείται και το 1:1 και το 1:N. Παρόλα αυτά, το σχήμα N:N έχει και το πρόβλημα του λυγισμού των log blocks αλλά και το πρόβλημα της υψηλής συσχέτισης των blocks.

4. Παραδοσιακοί αλγόριθμοι αντικατάστασης Buffer

Οι παραδοσιακοί αλγόριθμοι αντικατάστασης buffer έχουν σχεδιαστεί για συστήματα στα οποία η δευτερεύουσα μνήμη είναι ένας σκληρός δίσκος, ενώ κύριο μέλημα τους είναι η μεγιστοποίηση του hit ratio στην cache. Ωστόσο, συνήθως δεν είναι αποτελεσματικοί όταν χρησιμοποιηθούν σε υπολογιστικές συσκευές όπου η δευτερεύουσα μνήμη είναι μια μνήμη Flash, καθώς για την καλή απόδοση του συστήματος δεν αρκεί μόνο η μεγιστοποίηση του hit ratio, λόγω των ασύμμετρων καθυστερήσεων ανάγνωσης και εγγραφής της μνήμης Flash. Αν προσθέσουμε τις δαπανηρές λειτουργίες διαγραφής και τον περιορισμένο αριθμό διαγραφών για κάθε κελί της μνήμης Flash, συμπεραίνουμε ότι οι παραδοσιακοί αυτοί αλγόριθμοι δεν είναι αποδοτικοί για τη μνήμη Flash.

4.1 LRU (Least Recently Used)

Ο αλγόριθμος αντικατάστασης LRU (Least Recently Used) έχει χρησιμοποιηθεί ευρέως σε συστήματα με αποθήκευση δίσκου εξαιτίας της απλότητας του. Παρόλα αυτά, έχουν βρεθεί πολλές ανώμαλες συμπεριφορές για μερικά τυπικά workloads, όπου το hit ratio του LRU αυξάνεται ελάχιστα για μια μεγάλη αύξηση του μεγέθους της cache. Διάφορες παρατηρήσεις δείχνουν την ανικανότητα του LRU να χειριστεί patterns προσπελάσεων με μικρή τοπικότητα, όπως η σάρωση αρχείων, τακτές προσπελάσεις σε περισσότερα blocks από το μέγεθος της cache, και προσπελάσεις σε blocks με διακριτή συχνότητα. Ακολουθούν δύο αντιπροσωπευτικά παραδείγματα που δείχνουν τα μειονεκτήματα του LRU :

1). Μια έκρηξη αναφορών σε blocks τα οποία δεν χρησιμοποιούνται πολύ συχνά, όπως για παράδειγμα σε διαδοχικές σαρώσεις μεγάλων αρχείων, μπορεί να προκαλέσουν την αντικατάσταση των συχνά χρησιμοποιούμενων blocks από την cache. Δηλαδή, ο LRU αδυνατεί να προβλέψει την αντικατάσταση των ζεστών blocks από τα κρύα blocks.

2). Για ένα κυκλικό pattern αναφορών, όπως για παράδειγμα μια επαναληπτική προσπέλαση ενός αρχείου το οποίο είναι ελαφρώς μεγαλύτερο από το μέγεθος της cache, ο LRU λαθεμένα εκδιώκει από την cache τα blocks που πρόκειται να προσπελαστούν σύντομα, γιατί δεν έχουν προσπελαστεί για μεγαλύτερο διάστημα από τα υπόλοιπα. Δηλαδή, ο LRU αδυνατεί να διατηρήσει ένα ποσοστό επιτυχίας στην cache ανάλογο με το μέγεθος της.

4.2 Εισαγωγή στον LIRS

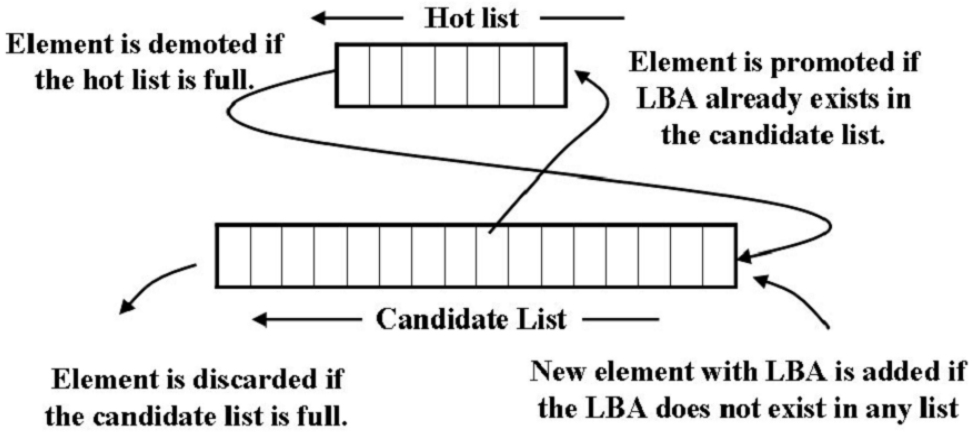
Ο αλγόριθμος αντικατάστασης LIRS (low inter-reference recency set) είναι ένας ενισχυμένος αλγόριθμος, ο οποίος χρησιμοποιεί ως μέτρο σύγκρισης, εκτός από το πόσο πρόσφατα προσπελάστηκε μια σελίδα (recency), και το πόσο συχνά χρησιμοποιείται μια σελίδα (frequency). Ο LIRS διατηρεί δύο στοίβες μεταβλητού μεγέθους και διαχωρίζει τις σελίδες σε σελίδες LIR και σελίδες HIR. Οι σελίδες LIR είναι αυτές που έχουν προσπελαστεί ξανά, ενώ παραμένουν στην cache, ενώ οι σελίδες HIR είναι αυτές που δεν βρίσκονταν στην cache όταν προσπελάστηκαν.

Ο LIRS χρησιμοποιεί το IRR (Inter-Reference Recency) μιας σελίδας, το οποίο αναφέρεται στον αριθμό των διακριτών προσπελάσεων άλλων σελίδων, μεταξύ δύο συνεχόμενων αναφορών στη συγκεκριμένη σελίδα. Επίσης, ο LIRS υποθέτει ότι αν το IRR μιας σελίδας είναι μεγάλο, τότε είναι πολύ πιθανό το επόμενο IRR της σελίδας να είναι επίσης μεγάλο. Με αυτήν την υπόθεση, επιλέγεται ως θύμα η σελίδα με το μεγαλύτερο IRR, διότι είναι πολύ πιθανό η συγκεκριμένη σελίδα να εκδιωχθεί αργότερα από τον LRU, προτού προσπελαστεί ξανά. Οι σελίδες LIR δεν επιλέγονται για αντικατάσταση, και δεν υπάρχουν σφάλματα σελίδας (page faults) για τις σελίδες αυτές. Μόνο ένα μικρό μέρος της cache χρησιμοποιείται για τις σελίδες HIR. Ουσιαστικά, ο LIRS επιλέγει ως θύμα τη σελίδα HIR με τη μεγαλύτερη τιμή recency ανάμεσα στις υπόλοιπες σελίδες HIR. Όμως, όταν μια σελίδα LIR αποκτήσει μεγάλο recency και η τιμή του recency μιας σελίδας HIR ελαττωθεί και γίνει μικρότερη από την αντίστοιχη της σελίδας LIR, τότε οι σελίδες αυτές εναλλάσσονται, δηλαδή η σελίδα LIR γίνεται HIR ενώ η HIR γίνεται LIR.

Ο LIRS συνήθως υπερισχύει του LRU επειδή εμφανίζει πολύ καλή απόδοση για κυκλικά pattern αναφορών, στα οποία ο LRU αδυνατεί να αντεπεξέλθει. Ωστόσο, μερικές φορές ο LIRS παρουσιάζει φτωχότερες επιδόσεις από τον LRU όταν το μέγεθος της cache είναι μεγαλύτερο από το πλήθος των χρησιμοποιούμενων σελίδων. Ένα άλλο μειονέκτημα του LIRS είναι ότι απαιτεί περισσότερο χώρο μνήμης, καθώς τα metadata των ήδη εκδιωγμένων σελίδων παραμένουν στη στοίβα που διατηρεί τις σελίδες LIR. Παρόλα αυτά ο LIRS παρουσιάζει καλύτερα αποτελέσματα κατά την προσομοίωση, σε σχέση με τον LRU, και για αυτό το λόγο επιλέχθηκε ως αλγόριθμος βάσης, ο οποίος θα ενισχυθεί για να γίνει αποτελεσματικός και σε συστήματα τα οποία έχουν ως δευτερεύουσα μνήμη, τη μνήμη Flash. Ο LIRS θα παρουσιαστεί λεπτομερώς παρακάτω ...

4.3 ARC (Adaptive Replacement Cache)

Τέλος, ο ARC (Adaptive Replacement Cache) είναι ένας άλλος παραδοσιακός αλγόριθμος αντικατάστασης buffer, ο οποίος υπερνικάει τις περισσότερες φορές τον LRU. Ο ARC επίσης διατηρεί δύο λίστες μεταβλητού μεγέθους, όπου κρατάει όχι μόνο τις σελίδες που βρίσκονται στην cache, αλλά και τα ίχνη των σελίδων που έχουν αντικατασταθεί. Η πρώτη LRU λίστα περιέχει τις κρύες σελίδες, οι οποίες έχουν προσπελαστεί μόνο μια φορά, ενώ η δεύτερη LRU λίστα περιέχει τις ζεστές σελίδες, οι οποίες έχουν προσπελαστεί τουλάχιστον δύο φορές πρόσφατα. Το μέγεθος της cache που απασχολεί κάθε λίστα αλλάζει ανάλογα με τον αριθμό των σφαλμάτων που συμβαίνουν σε κάθε λίστα. Δηλαδή, όταν συμβεί ένα σφάλμα σε μια λίστα, το μέγεθος της μειώνεται κατά μία θέση, ενώ παράλληλα το μέγεθος της άλλης αυξάνεται κατά μία θέση. Ο ARC είναι ένας αλγόριθμος χαμηλού κόστους και προσαρμόζεται στην αλλαγή του pattern αναφορών. Ωστόσο, στην περίπτωση που το μέγεθος της cache είναι ελάχιστα μικρότερο από το μέγεθος του συνόλου εργασίας (working set), έχουμε μια έκρηξη από σφάλματα σελίδας καθώς οι ζεστές σελίδες παραμένουν στην cache μολονότι δεν χρησιμοποιούνται πια.



Εικόνα 4.1 : Οι δύο LRU λίστες μεταβλητού μεγέθους του ARC

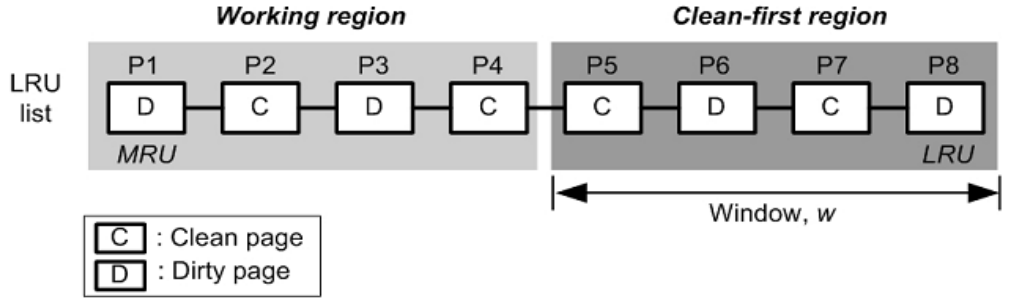
5. Αλγόριθμοι αντικατάστασης Buffer για μνήμη Flash

5.1 CFLRU (Clean First Least Recently Used)

Το κόστος αντικατάστασης μιας σελίδας που εκδιώκεται από τον buffer cache και μεταφέρεται στη μνήμη Flash μπορεί να ελαχιστοποιηθεί αν η σελίδα που επιλεχθεί είναι καθαρή. Μια καθαρή σελίδα περιέχει το ίδιο αντίγραφο δεδομένων με τα αυθεντικά δεδομένα που υπάρχουν στη μνήμη Flash, οπότε μπορεί απλώς να αφαιρεθεί από την cache, όταν εκδιώκεται από την πολιτική αντικατάστασης, χωρίς να προκαλέσει εγγραφή ή διαγραφή της μνήμης Flash.

Μια πολιτική αντικατάστασης του buffer cache μπορεί να αποφασίσει να κρατήσει στην cache όσο το δυνατόν περισσότερες βρόμικες σελίδες, για να γλιτώσει το κόστος εγγραφής τους στη μνήμη Flash. Ωστόσο, με την τακτική αυτήν, ο ελεύθερος χώρος της cache θα εξαντληθεί και το πλήθος των αποτυχιών/σφαλμάτων στην cache θα αυξηθεί δραματικά. Αυτό με τη σειρά του θα έχει ως αποτέλεσμα την αύξηση του κόστους αντικατάστασης, καθώς αυξάνεται ο αριθμός των αναγνώσεων από τη μνήμη Flash. Από την άλλη πλευρά, μια πολιτική αντικατάστασης που επικεντρώνεται στη μεγιστοποίηση του ποσοστού επιτυχίας (hit ratio) στην cache, θα εκδιώκει βρόμικες σελίδες, οι οποίες με τη σειρά τους θα αυξήσουν το κόστος αντικατάστασης καθώς εγγράφονται στη μνήμη Flash. Επομένως, μια σοφή πολιτική αντικατάστασης πρέπει να συμβιβάζεται και με τις δύο πλευρές προκειμένου να ελαχιστοποιήσει το συνολικό κόστος αντικατάστασης.

Ο CFLRU διαχωρίζει τη λίστα LRU σε δύο περιοχές για την εύρεση ενός σημείου ελαχίστου κόστους, όπως φαίνεται στην Εικόνα 5.1. Η περιοχή εργασίας (working region) αποτελείται από πρόσφατα χρησιμοποιημένες σελίδες και οι περισσότερες επιτυχίες (hits) στην cache παράγονται στην περιοχή αυτή. Η clean-first περιοχή αποτελείται από σελίδες που είναι υποψήφιες για έξωση. Ο CFLRU επιλέγει πρώτα μια καθαρή σελίδα για έξωση από την περιοχή clean-first για να γλιτώσει το κόστος εγγραφής της μνήμης Flash. Εάν δεν υπάρχει καθαρή σελίδα στην περιοχή αυτή, εκδιώκεται μια βρώμικη σελίδα στο τέλος της λίστας LRU. Για παράδειγμα, σύμφωνα με τον αλγόριθμο αντικατάστασης LRU, γίνεται έξωση στην τελευταία σελίδα της λίστας LRU. Έτσι, η προτεραιότητα να γίνει θύμα μια σελίδα στην Εικόνα 5.1 είναι P8, P7, P6 και P5. Ωστόσο, σύμφωνα με τον αλγόριθμο αντικατάστασης CFLRU, η προτεραιότητα είναι P7, P5, P8 και P6.



Εικόνα 5.1 : Παράδειγμα λειτουργίας του αλγορίθμου CFLRU

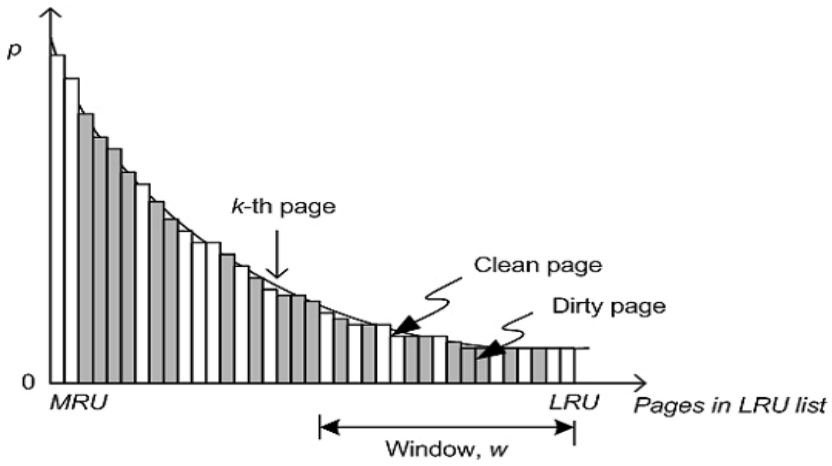
Το μέγεθος της clean-first περιοχής είναι το μέγεθος του παραθύρου, W. Η σωστή προσαρμογή του μεγέθους του παραθύρου είναι πολύ σημαντική, διότι το ποσοστό επιτυχίας (hit rate) μπορεί να μειωθεί δραματικά αν το μέγεθος του παραθύρου γίνει πάρα πολύ μεγάλο.

Η εύρεση του σωστού μεγέθους του παραθύρου της clean-first περιοχής είναι σημαντική για να ελαχιστοποιηθεί το συνολικό κόστος αντικατάστασης. Ένα μεγάλο παράθυρο θα αυξήσει το ποσοστό αποτυχίας στην cache, ενώ ένα μικρό μέγεθος παραθύρου θα αυξήσει τον αριθμό των εκδιωγμένων βρόμικων σελίδων, δηλαδή, τον αριθμό των εγγραφών στη μνήμη Flash. Οι λειτουργίες εγγραφής στη μνήμη Flash μπορούν επίσης να προκαλέσουν ένα μεγάλο αριθμό δαπανηρών διαγραφών.

Ας υποθέσουμε ότι CW είναι το κόστος μιας λειτουργίας εγγραφής στη μνήμη Flash και CR είναι το κόστος μιας λειτουργίας ανάγνωσης από τη μνήμη Flash. Αν ND είναι ο αριθμός των βρόμικων σελίδων που θα έπρεπε να είχαν εκδιωχθεί με τη σειρά LRU, αλλά παραμένουν στη μνήμη cache, το όφελος του αλγορίθμου CFLRU είναι CW •ND . Αν NC είναι ο αριθμός των καθαρών σελίδων που εκδιώχθηκαν αντί των βρόμικων σελίδων από την clean-first περιοχή, και Pi(k) είναι η πιθανότητα μελλοντικής αναφοράς της καθαρής σελίδας, i, η οποία εκδιώκεται στην k-οστή θέση, το κόστος του αλγορίθμου CFLRU μπορεί να υπολογιστεί από τη σχέση Σ CR • Pi(k) .

Η Εικόνα 5.2 δείχνει την πιθανότητα μελλοντικής αναφοράς για κάθε θέση της λίστας LRU. Σε αυτό το γράφημα, ο άξονας των x δείχνει τη θέση μιας σελίδας στη λίστα LRU. Το αριστερό άκρο του άξονα x είναι η πιο πρόσφατα χρησιμοποιημένη σελίδα και το δεξί άκρο του άξονα x είναι η λιγότερο πρόσφατα χρησιμοποιημένη σελίδα. Ο άξονας των y δείχνει την πιθανότητα μελλοντικής αναφοράς για κάθε σελίδα. Για παράδειγμα, αν η k-οστή σελίδα στη λίστα LRU έχει επιλεγεί για έξωση, είναι πιθανό να γίνει στο μέλλον αναφορά σε αυτήν και να προσκομιστεί ξανά στην cache με πιθανότητα P(k). Από ό,τι έχει συζητηθεί παραπάνω, ο τύπος (1) συνοψίζει το ορθό μέγεθος παραθύρου, W, της clean-first περιοχής :

$$\underset{w}{MAX}\left[C_W \bullet N_D - \sum_{i}^{N_C} C_R \bullet P_i(k)\right] \qquad (1)$$



Εικόνα 5.2 : Το μέγεθος παραθύρου του αλγορίθμου CFLRU

Εν τούτις, στην πραγματικότητα, δεν είναι εύκολο να βρούμε την πιθανότητα μελλοντικής αναφοράς για κάθε θέση της λίστας LRU. Η διερεύνηση του σωστού μέγεθος του παραθύρου της clean-first περιοχής γίνεται με δύο μεθόδους : την στατική και τη δυναμική. Η στατική μέθοδος αρχικά καθορίζει το μέγεθος του παραθύρου με μια καλή μέση τιμή, προερχόμενη από επαναλαμβανόμενα πειράματα σε ένα συγκεκριμένο σύνολο εφαρμογών. Ωστόσο, η στατική μέθοδος δεν μπορεί να προσαρμόσει δυναμικά το μέγεθος του παραθύρου σε διάφορα μεγέθη της cache και σε διαφορετικά σύνολα εφαρμογών. Η δυναμική μέθοδος μπορεί να προσαρμόσει κατάλληλα το μέγεθος του παραθύρου βασιζόμενη στη συλλογή πληροφοριών περιοδικά, σχετικά με τις λειτουργίες ανάγνωσης και εγγραφής στη μνήμη Flash. Αν vW και vR αντιπροσωπεύουν την αναλογία των εγγραφών και αναγνώσεων αντίστοιχα για μια δεδομένη χρονική περίοδο, η διαφορά κόστους μεταξύ γειτονικών περιόδων, Δ(vW· CW + vR· CR) μπορεί να ελέγξει την αύξηση ή μείωση του μεγέθους του παραθύρου.

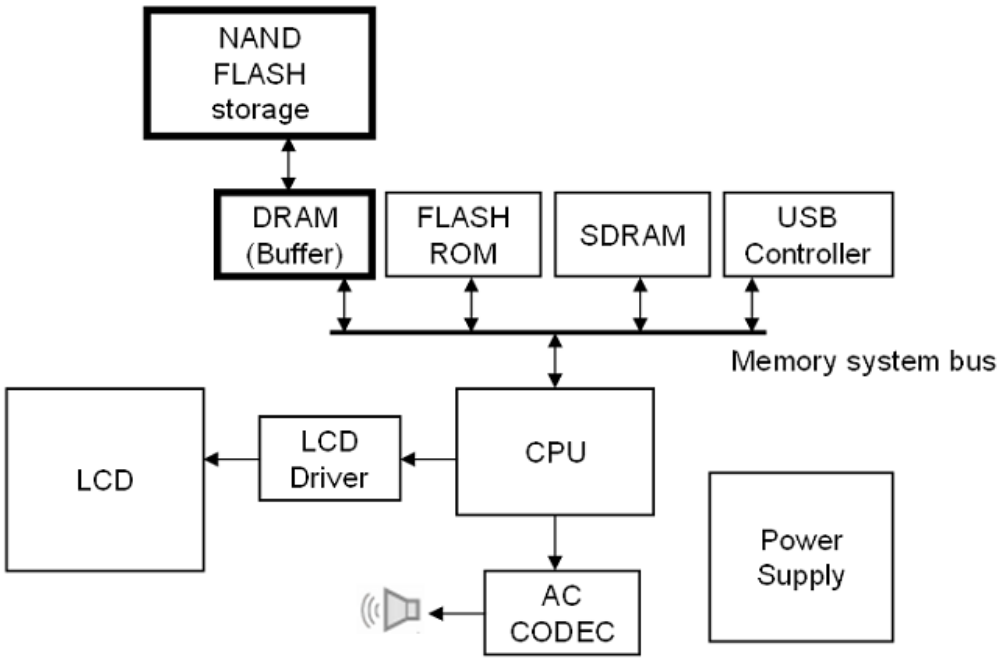
Το μεγάλο μειονέκτημα του CFLRU είναι η δύσκολη προσαρμογή του παραθύρου W σε εργασίες με διαφορετικά workloads. Επίσης, ο CFLRU πρέπει να γνωρίζει για κάθε σελίδα που βρίσκεται εντός του παραθύρου, αν είναι βρόμικη ή καθαρή. Πολλές φορές πραγματοποιεί περισσότερες αναγνώσεις της μνήμης Flash από τον LRU, πράγμα που μειώνει τη συνολική απόδοση. Τέλος, όταν εμφανίζονται μόνο αιτήσεις εγγραφής στη μνήμη Flash, ο CFLRU αδυνατεί να μειώσει το πλήθος των εγγραφών.

Ακολουθεί ο ψευδο-κώδικας του αλγορίθμου CFLRU :

```
read()
        check for data in cache
                IF fail
                        cache_CFLRU_write()
                        latency += flash_read_latency
                        update cache table
                                IF eviction
                                        disk_write()
                                        update LPN
Write()
        Update cache table
                IF there is empty space in cache
                        write new data in cache
                ELSE
                        WHILE (page in clean_first_region)
                        {
                                IF (Page = LRU && Page = clean)
                                        evict page_clean
                                        get plane_num
                                        exit while
                                ELSE look for the LRU dirty page
                                        evict page_dirty
                                        get plane_num
                        }
                write_cache()

                IF page = last page
                        allocate new block
```

5.2 FAB (Flash-Aware Buffer Management) Policy

Η Εικόνα 5.3 παρουσιάζει το συνολικό διάγραμμα αρχιτεκτονικής ενός τυπικού συστήματος PMP (Portable Media Player), όπως για παράδειγμα ενός MP3 ή MP4 player. Όλες οι αιτήσεις από τον επεξεργαστή (CPU) περνούν από τον DRAM buffer, ο οποίος εξυπηρετεί και αιτήσεις ανάγνωσης αλλά και αιτήσεις εγγραφής. Αν τα ζητούμενα δεδομένα δεν βρίσκονται στον buffer, προσκομίζονται από τη μνήμη Flash. Υποθέτουμε ότι η μνήμη Flash διαθέτει τον δικό της ελεγκτή (controller) για το FTL.



Εικόνα 5.3 : το συνολικό διάγραμμα αρχιτεκτονικής ενός PMP

Ο buffer σε μια συσκευή PMP συνήθως είναι πλήρως απασχολημένος μετά την εκκίνηση (boot) του συστήματος. Για να διασφαλιστεί μια θέση στον buffer για νέα δεδομένα, μια θέση θύμα θα πρέπει να επιλεγεί για να εκδιωχθεί από τον buffer. Όπως έχει προαναφερθεί, η πιο συνηθισμένη πολιτική αντικατάστασης είναι ο LRU, ο οποίος βασίζεται στη θεωρία ότι εάν μια θέση στη μνήμη έχει προσπελαστεί πρόσφατα, τότε θα προσπελαστεί ξανά στο κοντινό μέλλον. Παρόλα αυτά, ο LRU δεν αποτελεί την καλύτερη επιλογή για τη μνήμη Flash, εξαιτίας των ιδιαίτερων χαρακτηριστικών της. Για παράδειγμα, ένα τυπικό pattern προσβάσεων για ένα PMP αποτελείται από πολλές διαδοχικές μεγάλες προσβάσεις στα δεδομένα πολυμέσων (multimedia) και αρκετές αρκετές μικρές προσβάσεις στα μετα-δεδομένα (metadata) τους. Στην περίπτωση αυτή, ο LRU δεν μπορεί να διατηρήσει τα δεδομένα για τις σύντομες προσβάσεις στον buffer, γιατί οι μεγάλες διαδοχικές προσβάσεις τα διώχνουν από τον buffer.

Παρεμπιπτόντως, τα metadata μπορούν να ορισθούν χαλαρά ως δεδομένα που περιγράφουν δεδομένα και χρησιμοποιούνται για να περιγράφουν τη δομή, τον

ορισμό και τη διαχείριση των δεδομένων, με σκοπό την ευκολότερη μετέπειτα χρήση τους. Για παράδειγμα, μια ψηφιακή εικόνα μπορεί να περιέχει metadata που περιγράφουν το μέγεθος της εικόνας, το βάθος των χρωμάτων, την ανάλυση της, την ημερομηνία δημιουργίας της και άλλα.

Ο FAB ελαχιστοποιεί τον αριθμό των λειτουργιών εγγραφής και διαγραφής της μνήμης Flash, οι οποίες είναι οι πιο εξέχοντες πηγές καθυστέρησης. Επιπλέον, ο FAB βοηθάει τον ελεγκτή αποθήκευσης να χρησιμοποιήσει τα switch merges, τα οποία με τη σειρά τους μειώνουν το πλήθος των αντιγράφων των έγκυρων σελίδων κατά τη διαδικασία της συγχώνευσης. Επίσης, ο FAB μεγιστοποιεί την επανεγγραφή των ζεστών σελίδων, αφού επαναλαμβανόμενες εγγραφές σε ζεστές σελίδες έχουν ως αποτέλεσμα ένα μεγάλο αριθμό log blocks στη μνήμη Flash, τα οποία με τη σειρά τους προκαλούν τη συχνή συλλογή απορριμάτων (garbage collection). Έτσι, ελαχιστοποιείται ο χρόνος αναζήτησης των ζητούμενων δεδομένων στον buffer.

Αν ο buffer είναι πλήρης, ο FAB επιλέγει ένα block θύμα και εγγράφει όλες τις σελίδες που ανήκουν στο συγκεκριμένο block στη μνήμη Flash. Το block θύμα είναι εκείνο με τον μεγαλύτερο αριθμό σελίδων στον buffer. Η πολιτική του FAB είναι μια πολιτική block-level LRU, δηλαδή η λίστα του FAB είναι μια λίστα LRU που αποτελείται από blocks και όχι από σελίδες, προκειμένου να αποφύγει τα μειονεκτήματα της πολιτικής page-level LRU που αναφέρθηκαν προηγουμένως. Ταυτόχρονα, όταν εγγραφές συνεχόμενων δεδομένων γεμίζουν τον buffer, ο FAB επιλέγει τις σελίδες αυτές ως θύματα, με σκοπό να μεγιστοποιήσει την πιθανότητα να συμβεί ένα switch merge στη μνήμη Flash. Αυτό μεγιστοποιεί επίσης το ποσοστό επιτυχίας των ζεστών σελίδων, διότι οι σελίδες των σύντομων εγγραφών μπορούν να παραμείνουν για περισσότερο καιρό στον buffer.

Όταν ο FAB δεχθεί μια αίτηση εγγραφής, αρχικά ψάχνει στον buffer. Σε περίπτωση επιτυχίας, τα δεδομένα επανεγγράφονται στον buffer. Διαφορετικά, μια νέα θέση μνήμης δεσμεύεται στον buffer και τα δεδομένα εγγράφονται εκεί. Αφού όλες οι εγγραφές πραγματοποιούνται πρώτα στον buffer, οι πραγματικές εγγραφές στη μνήμη Flash συμβαίνουν μόνο κατά την αντικατάσταση ενός block.

Στην περίπτωση που ο buffer είναι πλήρης, επιλέγεται ως θύμα το block με τις περισσότερες σελίδες στον buffer. Το συγκεκριμένο block εκδιώκεται από τον buffer και μόνο οι βρόμικες σελίδες του εγγράφονται στη μνήμη Flash, καθώς οι καθαρές περιέχουν τα ίδια δεδομένα με αυτά που περιέχει η μνήμη Flash. Όταν υπάρχει ισοπαλία, δηλαδή υπάρχουν περισσότερα από ένα block στον buffer, τα οποία περιέχουν τον ίδιο αριθμό σελίδων, τότε επιλέγεται το λιγότερο πρόσφατα χρησιμοποιημένο block, όπως δηλαδή στον LRU. Η πολιτική αυτή μειώνει τον αριθμό των αντιγραφών σελίδων στη μνήμη Flash, κατά τη συγχώνευση των blocks. Αφού τα περισσότερα δεδομένα των PMP συσκευών είναι συνεχόμενα, η πλειονότητα των blocks της μνήμης Flash είναι είτε γεμάτα από έγκυρες σελίδες, είτε άδεια. Ωστόσο, για τα blocks που χρησιμοποιούνται από metadata, είναι πολύ πιθανό μόνο μερικές σελίδες να ενημερώνονται συχνά και μερικές έγκυρες και άκυρες σελίδες να αναμειγνύονται μαζί. Επιλέγοντας ως θύμα ένα γεμάτο από σελίδες block, ο FAB ευνοεί τα metadata έναντι των κανονικών δεδομένων, κάτι το οποίο είναι πολύ επιθυμητό στην περίπτωση των PMP συσκευών.

Η Εικόνα 5.4 δείχνει τον ψευδοκώδικα για τη λειτουργία εγγραφής.

```
FAB_Write (block, page, data)
{
   if ((bufloc = Search_Buffer (block, page)) != null) {
      Write_Page (bufloc, data);
   }
   else {
      if (Buffer_Full ()) {
         victim  = Select_Victim_Block ();
         Flush_Victim_Block (victim);
      }
      bufloc = Allocate_New_Page ();
      Write_Page (bufloc, data);
   }
   Rearrange_Blocklist_For_LRU (block);

}
```

Εικόνα 5.4 : ο χειρισμός μιας αίτησης εγγραφής από τον FAB

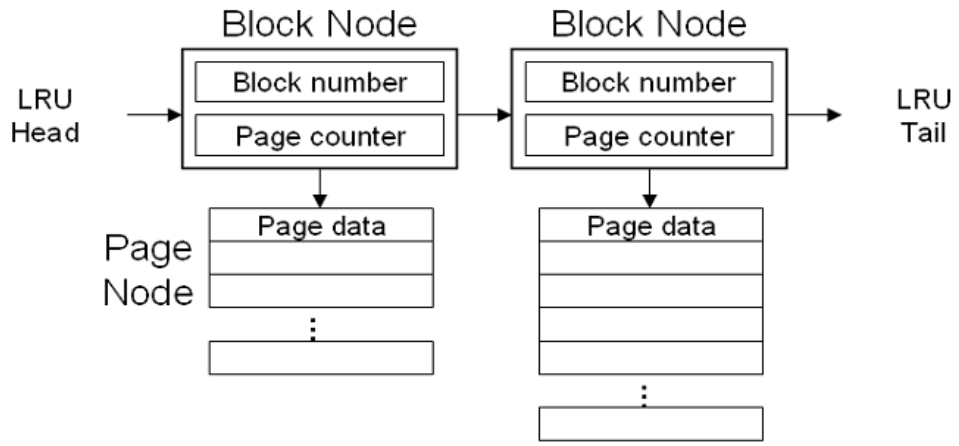
Για μια αίτηση ανάγνωσης, ο FAB λειτουργεί παρόμοια. Σε περίπτωση επιτυχίας στον buffer, ο FAB επιστρέφει αμέσως τα δεδομένα στον επεξεργαστή (CPU). Αν ο FAB δεν μπορέσει να βρει τα ζητούμενα δεδομένα στον buffer, διαβάζει τα δεδομένα από τη μνήμη Flash, τα αντιγράφει στον buffer και τα επιστρέφει στον επεξεργαστή. Στη συνέχεια, επιλέγεται ένα block ως θύμα και ο FAB εγγράφει όλες τις καθαρές σελίδες που ανήκουν στο συγκεκριμένο block, στη μνήμη Flash.

Στην Εικόνα 5.5 φαίνεται ο ψευδοκώδικας της λειτουργίας ανάγνωσης υπό τον FAB.

```
FAB_Read (block, page, data)
{
   if ((bufloc = Search_Buffer (block, page)) != null) {
      Rearrange_Blocklist_For_LRU (block);
      Read_Page (bufloc, data);
   }
   else {
      if (Buffer_Full ()) {
         victim  = Select_Victim_Block ();
         Flush_Victim_Block (victim);
      }
      bufloc = Allocate_New_Page ();
      Read_From_Flash (bufloc, block, page);
      Read_Page (bufloc, data);
   }
   Rearrange_Blocklist_For_LRU (block);
}
```

Εικόνα 5.5 : ο χειρισμός μιας αίτησης ανάγνωσης από τον FAB

Για να ελαχιστοποιηθεί ο χρόνος αναζήτησης στον buffer, οι σχεδιαστές του FAB χρησιμοποίησαν μια δομή δεδομένων όπως απεικονίζεται στην Εικόνα 5.6. Η block node list είναι η λίστα των blocks είναι ταξινομημένη κατά recency, δηλαδή κατά πόσο πρόσφατα χρησιμοποιήθηκε κάθε block, ενώ η page node list είναι η λίστα των σελίδων που ανήκουν στο αντίστοιχο block.



Εικόνα 5.6 : η δομή δεδομένων του αλγορίθμου FAB

Ένας κόμβος στη λίστα των blocks διαθέτει ένα αριθμό block, ένα μετρητή σελίδων, ένα δείκτη προς τον επόμενο κόμβο στη λίστα των blocks και ένα δείκτη προς τη λίστα των σελίδων που ανήκουν στο συγκεκριμένο block. Ο αριθμός block ταυτοποιεί ένα μοναδικό block της μνήμης Flash. Ο μετρητής σελίδων υποδηλώνει τον αριθμό των σελίδων που διαθέτει κάθε block και αποτελεί το κύριο κριτήριο κατά την επιλογή ενός block ως θύμα.

Κατά τη διαδικασία της αναζήτησης στον buffer, το block που περιέχει τη ζητούμενη σελίδα μπορεί να βρεθεί διαιρώντας τον αριθμό της σελίδας με το μέγεθος του block. Στη συνέχεια, ο FAB αναζητεί το συγκεκριμένο block στην αντίστοιχη λίστα των blocks. Παρόλο που υλοποιείται με διαδοχικές αναζητήσεις και έχει πολυπλοκότητα O(n), το πραγματικό κόστος δεν είναι τόσο υψηλό καθώς ο αριθμός των κόμβων που μπορούν να υπάρχουν στη λίστα των blocks περιορίζεται από το μέγεθος του buffer. Πιο πολύπλοκες δομές δεδομένων όπως οι πίνακες κατακερματισμού και τα ζυγισμένα δέντρα μπορούν να υιοθετηθούν για πιο γρήγορη αναζήτηση, αλλά δυσκολεύουν την υλοποίηση του μηχανισμού του LRU χωρίς την κατασκευή μιας επιπρόσθετης λίστας LRU. Ούτως ή άλλως, σύμφωνα με τις μετρήσεις που πραγματοποίησαν οι εισηγητές του FAB, αυτή η απλή γραμμική αναζήτηση δεν προκαλεί κάποιο αξιοσημείωτο κόστος.

Αν το block που περιέχει τα ζητούμενα δεδομένα δεν βρεθεί στον buffer, ο FAB τερματίζει αμέσως την αναζήτηση αφού έχει συμβεί σφάλμα στον buffer. Αν όμως το block βρεθεί, ο FAB συνεχίζει την αναζήτηση στη λίστα των σελίδων. Η λίστα αυτή είναι δομημένη ως ένα δέντρο αναζήτησης για αποτελεσματικές αναζητήσεις. Αφού βρεθεί ο αντίστοιχος κόμβος-σελίδα, ο FAB επαναταξινομεί τη λίστα LRU μεταφέροντας τον κόμβο-block που περιέχει τη ζητούμενη σελίδα στην κεφαλή της λίστας των blocks, δηλαδή στη θέση MRU της λίστας.

Όταν η ζητούμενη σελίδα δεν βρεθεί στον buffer, μια νέα σελίδα προστίθεται στη λίστα των σελίδων του αντίστοιχου block. Ο μετρητής σελίδων του συγκεκριμένου block αυξάνεται κατά ένα. Αν ο κόμβος-block δεν υπάρχει στον buffer, ένας νέος κόμβος-block πρέπει να προστεθεί στην κεφαλή της λίστας των κόμβων και ο μετρητής των σελίδων του πρέπει να τεθεί ίσος με ένα.

Αν η αποτυχία προέρχεται από μια αίτηση ανάγνωσης, η ζητούμενη σελίδα προσκομίζεται από τη μνήμη Flash και αποθηκεύεται στη νέα σελίδα που μόλις προστέθηκε στον buffer. Αν προέρχεται από αίτηση εγγραφής, τα δεδομένα εγγράφονται μόνο στη νέα σελίδα, αφήνοντας ανέπαφα τα παλιά δεδομένα στη μνήμη Flash. Αφού οι επόμενες αιτήσεις για τη συγκεκριμένη σελίδα θα εξυπηρετηθούν από τον buffer, το παλιό αντίγραφο της σελίδας στη μνήμη Flash δεν προκαλεί κανένα πρόβλημα.

Όταν γεμίσει ο buffer, πρέπει να εκδιωχθούν ένας αριθμός σελίδων. Αν η σελίδα είναι βρόμικη, δηλαδή έχει τροποποιηθεί για όσο βρισκόταν στον buffer, τότε εγγράφεται στη μνήμη Flash, ενώ οι καθαρές σελίδες απλώς αφαιρούνται από τον buffer χωρίς να ενημερώνονται τα αντίγραφα τους στη μνήμη Flash. Ο FAB αφού είναι ένα σχήμα block-level LRU, επιλέγει κάθε φορά ένα ολόκληρο block, αντί για μία μονάχα σελίδα με βάση τα κριτήρια που έχουν προαναφερθεί προηγουμένως.

Ο FAB αρχίζει την αναζήτηση του block θύματος από την ουρά της λίστας των κόμβων-blocks και αν βρει κάποιο block, το οποίο είναι γεμάτο από σελίδες, σταματάει την αναζήτηση και το επιλέγει ως θύμα. Αυτή η περίπτωση είναι και η βέλτιστη. Θεωρώντας τα χαρακτηριστικά των αρχείων που χρησιμοποιούνται στις συσκευές PMP, η λίστα των blocks υποτίθεται ότι πρέπει να περιέχει πολλά τέτοια blocks. Τα συγκεκριμένα blocks είναι γεμάτα από έγκυρες σελίδες, οπότε όταν εγγράφεται στη μνήμη Flash, το νέο block παίρνει τη θέση του παλιού (switch merge) και έτσι ελαχιστοποιούνται οι αντιγραφές έγκυρων σελίδων από το παλιό block.

Η Εικόνα 5.7 δείχνει τον ψευδοκώδικα για την επιλογή ενός block θύματος.

```
#define BlockPerPageNum 64
Select_Victim_Block ()
{
   VictimBlock = -1;
   MaxPageNum = 0;
   For each BlockNode from BlockNodeListTail to
   BlockNodeListHead
   {
      if (BlockNode.PageCount == BlockPerPageNum)
         return BlockNode.BlockNum;
      if (BlockNode.PageCount > MaxPageNum)
      {
         MaxPageNum = BlockNode.PageCount;
         VictimBlock = BlockNode.BlockNum;
      }
   }
   return VictimBlock;
}
```

Εικόνα 5.7: η επιλογή ενός block θύματος από τον FAB

Ο FAB επιδεικνύει καλή απόδοση όταν οι περισσότερες αιτήσεις Εισόδου/Εξόδου είναι διαδοχικές, όπως στην περίπτωση των συσκευών PMP. Ωστόσο μπορεί να ελαττώσει το ποσοστό επιτυχίας στην buffer cache όταν το pattern των αιτήσεων Εισόδου/Εξόδου είναι τυχαίο (random), με συνέπεια να βλάψει τη συνολική απόδοση του συστήματος. Επίσης, επιλέγοντας ως θύμα το block με τον μεγαλύτερο αριθμό σελίδων στον buffer, το οποίο μπορεί να βρίσκεται ακόμα και στη MRU θέση της λίστας LRU, ο FAB βάζει σε δεύτερη μοίρα το πόσο πρόσφατα χρησιμοποιήθηκαν οι σελίδες του block θύματος, με αποτέλεσμα να μειώνεται το ποσοστό επιτυχίας στην buffer cache. Τέλος, αν μία μόνο σελίδα ενός block έχει προσπελαστεί πρόσφατα, τότε όλες οι υπόλοιπες σελίδες που ανήκουν στο ίδιο block παραμένουν στην buffer cache, ακόμα και αν δεν έχουν χρησιμοποιηθεί πρόσφατα. Επιπλέον, με την αναφορά σε μία μόνο σελίδα του block, όλες οι υπόλοιπες μεταφέρονται στην κεφαλή της λίστας του LRU, δηλαδή στη θέση MRU. Έτσι όμως, σπαταλάται πολύτιμος χώρος από τον buffer για τις μη πρόσφατα χρησιμοποιημένες σελίδες.

5.3 LRU – WSR (Write Sequence Reordering)

Ο στόχος του αλγορίθμου LRU-WSR είναι η μείωση του αριθμού των εγγραφών των βρόμικων σελίδων στη μνήμη Flash όταν συμβαίνει αντικατάστασης σελίδας. Για να επιτύχει αυτόν τον στόχο, χρησιμοποιεί την ακόλουθη στρατηγική : καθυστερεί όσο το δυνατό περισσότερο την εκδίωξη των σελίδων που είναι βρόμικες και έχουν μεγάλη συχνότητα πρόσβασης. Χρησιμοποιώντας τη στρατηγική αυτή, το ποσοστό επιτυχίας στην cache χρησιμοποιώντας τον LRU-WSR μπορεί να είναι μικρότερο από εκείνο του LRU, με αποτέλεσμα να αυξάνεται το πλήθος των αναγνώσεων από τη μνήμη Flash. Παρόλα αυτά, ο αλγόριθμος αυτός μειώνει σημαντικά τον αριθμό των εγγραφών των σελίδων στη μνήμη Flash, αλλά και το πλήθος των διαγραφών της. Ως αποτέλεσμα, αυξάνει τη συνολική απόδοση του συστήματος με δευτερεύουσα αποθήκευση τη μνήμη Flash.

Ο συνολικός χρόνος εκτέλεσης μπορεί να δοθεί από τον τύπο :

$$Runtime = a \times RC + b \times WC + c \times EC + C \quad (1)$$

όπου a: missed read count, b: flushed write count, c: erase count
RC: read cost, WC: write cost, EC: erase cost, C : CPU cost

Να σημειωθεί ότι μια λειτουργία ανάγνωσης της μνήμης Flash πραγματοποιείται όταν συμβαίνει μια αποτυχία στον buffer, και μια λειτουργία εγγραφής στη μνήμη Flash πραγματοποιείται όταν μόνο όταν μια βρόμικη σελίδα επιλέγεται ως θύμα από την πολιτική αντικατάστασης. Το πλήθος των διαγραφών δεν μπορεί να υπολογιστεί στο επίπεδο του buffer, αλλά γενικά είναι ανάλογο με το πλήθος των εγγραφών στη μνήμη Flash. Επομένως ο τύπος (1) μπορεί να μετατραπεί στον παρακάτω τύπο :

$$Runtime = a \times RC + b \times WC' + C \quad (2)$$

όπου a: read count, b: write count
RC: read cost, WC': write cost consider erase cost, C : CPU cost

Από τον τύπο (2) και τον πίνακα 1 (χαρακτηριστικά της μνήμης Flash) φαίνεται ότι το κόστος εγγραφής είναι πολύ μεγαλύτερο από το κόστος ανάγνωσης. Ο ακριβής λόγος των δύο αυτών καθυστερήσεων εξαρτάται από τον εκάστοτε αλγόριθμο FTL που χρησιμοποιείται σε κάθε συσκευή, μολονότι ο αλγόριθμος FTL κρύβεται και έτσι μπορεί να βρεθεί μόνο μέσω πειραμάτων. Για παράδειγμα, το κόστος εγγραφής σε μια κάρτα μνήμης mini SD είναι 200 φορές μεγαλύτερο από το κόστος ανάγνωσης. Από το αποτέλεσμα αυτό, μπορεί εύκολα κανείς να μαντέψει ότι η απόδοση της μνήμης Flash θα αυξηθεί αν ελαττωθεί το πλήθος των εγγραφών σε αυτήν. Για τον σκοπό αυτό, η καθυστέρηση εγγραφής των βρόμικων σελίδων στη μνήμη Flash μπορεί να μειώσει τον αριθμό των εγγραφών. Ωστόσο, η πολιτική αυτή ενδέχεται να αυξήσει το πλήθος των αναγνώσεων εξαιτίας του υποβαθμισμένου ποσοστού επιτυχίας που προκαλεί. Το όφελος της μείωσης του αριθμού των εγγραφών στη μνήμη Flash πρέπει να είναι μεγαλύτερο από τη ζημιά που προκαλεί η αύξηση του πλήθους των αναγνώσεων, ούτως ώστε η συνολική απόδοση του συστήματος αποθήκευσης να βελτιωθεί. Με άλλα λόγια πρέπει να ισχύει ο παρακάτω τύπος :

$$b' \times WC' - a' \times RC > 0 \quad (3)$$

όπου a': increased read count, b: reduced write count

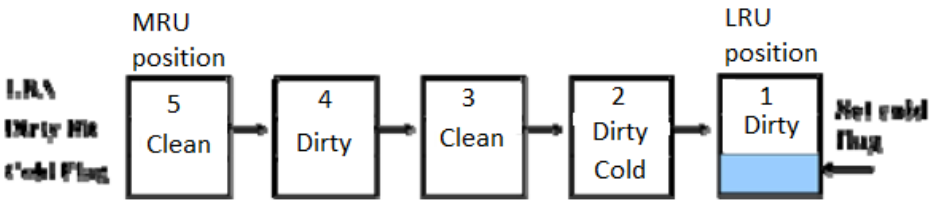
Για να αποφευχθεί η υποβάθμιση του ποσοστού επιτυχίας στην cache, χρησιμοποιείται η πολιτική του WSR (Write Sequence Reordering). Το βασικό σχήμα του WSR είναι το ακόλουθο :

1). Η χρησιμοποίηση αλγορίθμου που να μπορεί να κρίνει αν μια σελίδα είναι κρύα ή ζεστή
2). Η καθυστέρηση εγγραφής των σελίδων που είναι βρόμικες και ζεστές.

Για την υλοποίηση του WSR, μόνο ένα πεδίο που ονομάζεται cold-flag απαιτείται, για να μπορεί να κρίνει ο αλγόριθμος αν μια σελίδα είναι κρύα ή ζεστή. Όταν η πολιτική αντικατάστασης επιλέγει μια σελίδα ως θύμα, εξετάζεται αν η σελίδα είναι βρόμικη. Αν η σελίδα είναι βρόμικη και δεν έχει ενεργοποιηθεί το πεδίο cold-flag, η σελίδα θεωρείται ως ζεστή βρόμικη σελίδα. Τότε, ενεργοποιείται το πεδίο cold-flag της σελίδας και η πολιτική αντικατάστασης προσπαθεί να επιλέξει μια άλλη σελίδα ως θύμα. Αν η υποψήφια σελίδα είναι καθαρή ή κρύα και βρόμικη, τότε επιλέγεται ως θύμα και εκδιώκεται από την cache. Επιπρόσθετα, το πεδίο cold-flag μιας βρόμικης σελίδας απενεργοποιείται όταν η σελίδα προσπελάζεται ξανά και επομένως θεωρείται ζεστή.

Ο WSR είναι ένας ευρετικός αλγόριθμος που βασίζεται στον αλγόριθμο δεύτερης ευκαιρίας. Είναι πολύ δύσκολο να βρεθεί η ακολουθία του buffer που μεγιστοποιεί το αριστερό μέλος του τύπου (3), αλλά παρόλα αυτά, είναι αποδεδειγμένο μέσω πειραμάτων ότι ο WSR μειώνει αποτελεσματικά τον αριθμό των εγγραφών και διαγραφών της μνήμης Flash, χωρίς να επιφέρει σημαντική υποβάθμιση του ποσοστού επιτυχίας στον buffer.

Ο LRU-WSR προσπαθεί να μη διατηρεί τις κρύες και ταυτόχρονα βρόμικες σελίδες στον buffer, οι οποίες είναι λιγότερο πιθανό να προσπελαστούν ξανά σύντομα. Όπως δείχνει η Εικόνα 5.8, ο LRU-WSR χρησιμοποιεί μια λίστα από σελίδες και ένα επιπλέον πεδίο, το cold-flag. Όταν μια βρόμικη σελίδα επιλέγεται ως υποψήφιο θύμα, ελέγχεται το πεδίο της, cold-flag. Αν δεν έχει ενεργοποιηθεί, δηλαδή η σελίδα θεωρείται ζεστή, η σελίδα μεταφέρεται στη θέση MRU της λίστας, ενώ ενεργοποιείται το πεδίο της, cold-flag. Με άλλα λόγια η σελίδα γίνεται κρύα. Στη συνέχεια, μια άλλη υποψήφια σελίδα επιλέγεται για να εξεταστεί από τη θέση LRU της λίστας του buffer. Αν η υποψήφια σελίδα είναι βρόμικη και κρύα, η σελίδα εγγράφεται στη μνήμη Flash για να αποφευχθεί η μεγάλη μείωση του ποσοστού επιτυχίας στον buffer. Αν η σελίδα είναι καθαρή, επιλέγεται ως θύμα ανεξάρτητα αν η σελίδα είναι ζεστή ή κρύα. Τέλος όταν προσπελάζεται μια βρόμικη σελίδα στον buffer, η σελίδα μεταφέρεται στη θέση MRU της λίστας και απενεργοποιείται το πεδίο της, cold-flag, δηλαδή η σελίδα γίνεται ζεστή.



Εικόνα 5.8 : η LRU λίστα του αλγορίθμου LRU-WSR

Στην Εικόνα 5.9 παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου LRU-WSR. Αφού χρησιμοποιεί επιπλέον μόνο μια δυαδική μεταβλητή, το πεδίο cold-flag, το επιπλέον κόστος για τη δομή δεδομένων του αλγορίθμου είναι μηδαμινό. Τα υπόλοιπα μέρη του αλγορίθμου δεν διαφέρουν από τον αυθεντικό LRU.

```
L = buffer list of LRU
victim  = the page at LRU position in L
while (victim is dirty)
:     if (cold-flag of victim is set)
        exit while
      else
        move victim to MRU position in L
        set cold-flag of victim
        victim = the page at LRU position in L
   remove victim from L
return victim
```

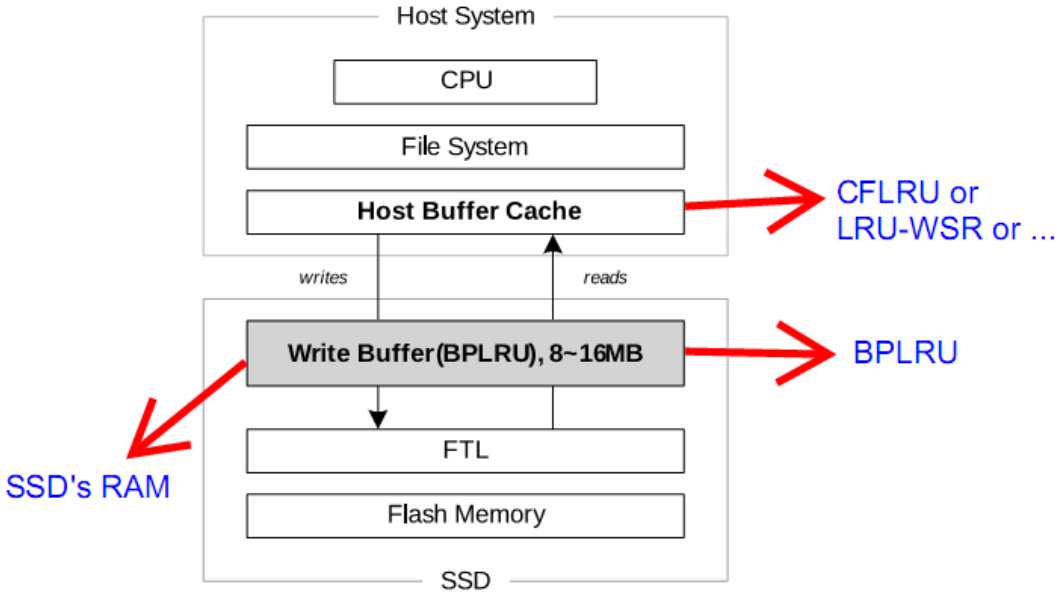
Εικόνα 5.9 : η επιλογή μιας σελίδας θύματος από τον LRU-WSR

Το κύριο μειονέκτημα του LRU-WSR, είναι ότι όταν η τοπικότητα στις εγγραφές (write locality) είναι χαμηλή, η πολιτική WSR μπορεί να μην είναι αποτελεσματική, και να μειώσει τη συνολική απόδοση. Αυτό οφείλεται στο γεγονός ότι το όφελος της μείωσης του αριθμού των εγγραφών στη μνήμη Flash γίνεται μικρότερο από το επιπλέον κόστος που παράγεται από την αύξηση του πλήθους των αναγνώσεων, εξαιτίας της υποβάθμισης του ποσοστού επιτυχίας στην buffer cache.

5.4 BPLRU (Block Padding LRU)

Ο μηχανισμός BPLRU σχεδιάστηκε για να εφαρμοστεί στον buffer εσωτερικά των δίσκων Solid-State (SSD), και όχι στην buffer cache (RAM) του συστήματος, όπως δηλαδή πράττουν οι προηγούμενοι αλγόριθμοι. Ο BPLRU κατανέμει και διαχειρίζεται τον buffer μέσα στον SSD μόνο για τις αιτήσεις εγγραφής. Για τις αιτήσεις ανάγνωσης, απλώς ανακατευθύνει τις αιτήσεις στο FTL. Οι σχεδιαστές του BPLRU επέλεξαν να χρησιμοποιήσουν όλη τη διαθέσιμη μνήμη RAM μέσα στον SSD ως buffer εγγραφών διότι τα περισσότερα συστήματα desktop και server διαθέτουν πολύ μεγαλύτερη μνήμη cache (RAM), η οποία μπορεί να απορροφήσει επαναλαμβανόμενες αιτήσεις ανάγνωσης για το ίδιο block πιο αποτελεσματικά, σε σχέση με την περιορισμένη μνήμη RAM που υπάρχει στο εσωτερικό των SSD.

Η Εικόνα 5.10 δείχνει τη γενική διαμόρφωση του συστήματος. Ο host (desktop, laptop, server, ...) διαθέτει έναν επεξεργαστή, ένα σύστημα αρχείων και μια buffer cache (RAM). Ο δίσκος Solid-State περιλαμβάνει τον buffer εγγραφών (RAM), το FTL, και βεβαίως τη μνήμη Flash. Για τον buffer του host, ένας αλγόριθμος αντικατάστασης για μνήμη Flash, όπως για παράδειγμα ο CFLRU ή ο LRU-WSR, μπορεί να εφαρμοστεί για να μειώσει τον αριθμό των αιτήσεων εγγραφών στη μνήμη Flash.



Εικόνα 5.10 : η διαμόρφωση του συστήματος. Ο BPLRU εφαρμόζεται στον RAM buffer εσωτερικά του δίσκου Solid-State.

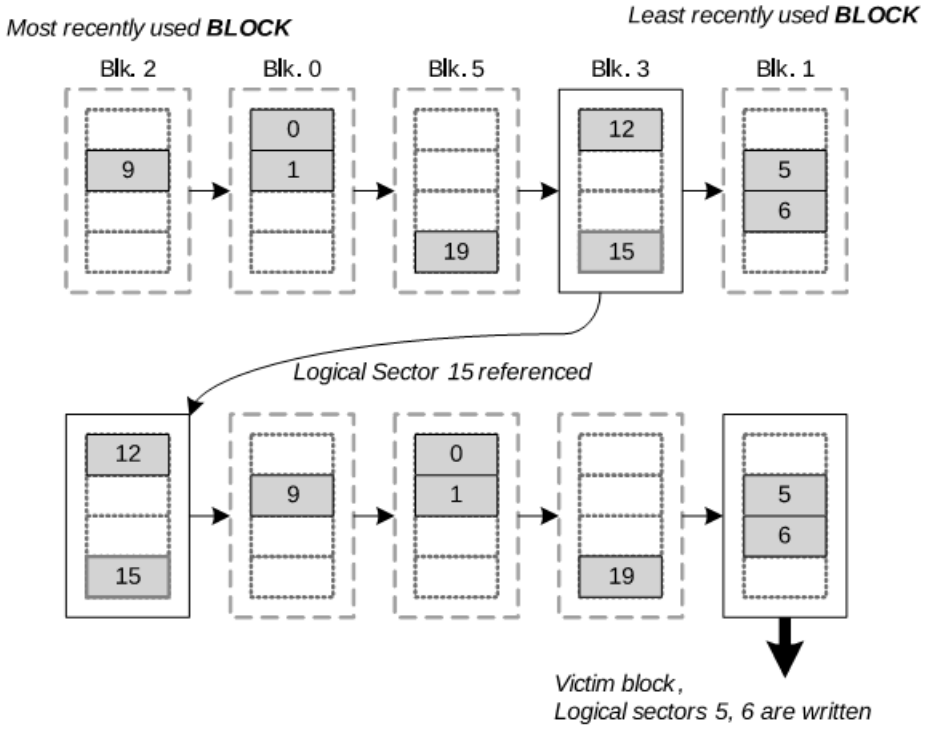
Ο BPLRU συνδυάζει τρεις τεχνικές : τη διαχείριση block-level LRU, το page padding και το LRU Compensation.

5.4.1. Block-level LRU

Ο BPLRU διαχειρίζεται τι λίστα LRU σε μονάδες των blocks. Όλα τα τμήματα του RAM buffer ομαδοποιούνται σε blocks, τα οποία έχουν το ίδιο μέγεθος με τα blocks

της μνήμης Flash. Όταν μια αποθηκευμένη σελίδα στον RAM buffer προσπελάζεται, όλες οι σελίδες που ανήκουν στο ίδιο block με την προσπελαζόμενη σελίδα μεταφέρονται στην κορυφή της λίστας LRU. Για να δημιουργηθεί ελεύθερος χώρος στον buffer, ο BPLRU επιλέγει το λιγότερο πρόσφατα χρησιμοποιημένο block, αντί της λιγότερο πρόσφατα χρησιμοποιημένης σελίδας, και εγγράφει όλες τις σελίδες του block θύματος στη μνήμη Flash. Η εγγραφή αυτή σε επίπεδο block, ελαχιστοποιεί το κόστος στο log-block FTL.

Η Εικόνα 5.11 παρουσιάζει ένα παράδειγμα της λίστας του BPLRU. Όπως φαίνεται παρακάτω, 8 σελίδες βρίσκονται στον buffer εγγραφών και κάθε block αποτελείται από 4 σελίδες. Όταν η σελίδα 15 επανεγγράφεται, τότε ολόκληρο το block που περιέχει τη συγκεκριμένη σελίδα, μεταφέρεται στην κεφαλή της λίστας LRU. Επομένως, η σελίδα 12 μεταφέρεται στην κεφαλή της λίστας, παρόλο που δεν έχει προσπελαστεί πρόσφατα. Όταν απαιτείται ελεύθερος χώρος, το λιγότερο χρησιμοποιημένο block επιλέγεται ως θύμα και όλες οι σελίδες που περιέχει απομακρύνονται από την cache. Στο συγκεκριμένο παράδειγμα, το block 1 επιλέγεται ως θύμα και οι σελίδες 5 και 6 εγγράφονται στη μνήμη Flash.



Εικόνα 5.11 : Ένα παράδειγμα του Block-level LRU. Όταν η σελίδα 15 προσπελάζεται, η σελίδα 12 μεταφέρεται στην κεφαλή της λίστας, παρόλο που δεν έχει προσπελαστεί πρόσφατα.

Αν οι αιτήσεις εγγραφών είναι τυχαίες, το ποσοστό επιτυχίας στην cache θα μειωθεί. Τότε, η LRU cache θα δράσει όπως μια ουρά FIFO και δεν θα επιρρεάσει την απόδοση. Ωστόσο, ο BPLRU μπορεί να βελτιώσει την απόδοση όσον αφορά τις εγγραφές, ακόμα και αν και δεν υπάρχουν καθόλου επιτυχίες στην cache. Αν υπολογίσουμε το ποσοστό επιτυχίας στην cache σε επίπεδο block, μπορεί να είναι μεγαλύτερο του μηδενός, ακόμα και αν το αντίστοιχο σε επίπεδο σελίδας είναι μηδέν.

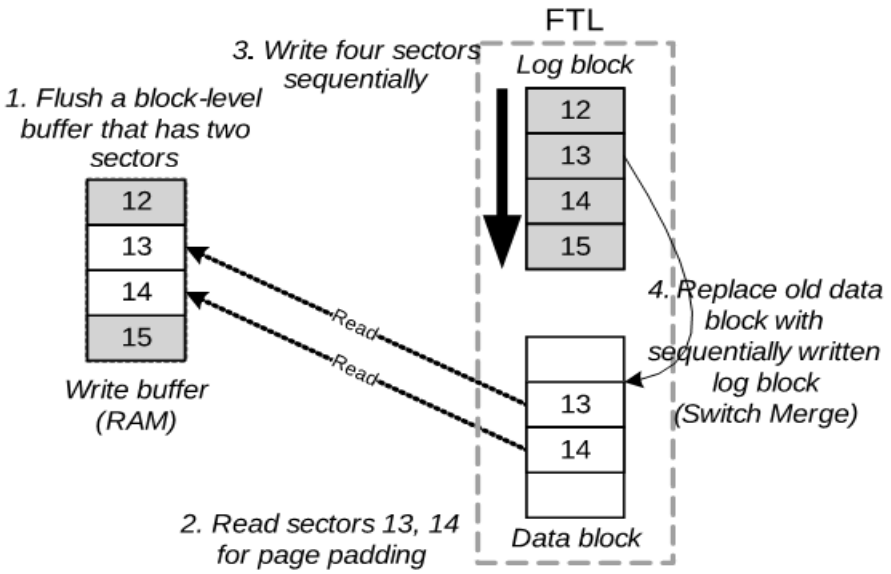
Ο BPLRU ελαχιστοποιεί το κόστος που παράγεται από τις συγχωνεύσεις των block, επαναδιατάσσοντας την ακολουθία εγγραφών για τη μνήμη Flash.

5.4.2 Page Padding

Ο BPLRU χρησιμοποιεί μια τεχνική page padding για ένα block θύμα, ούτως ώστε να ελαχιστοποιήσει το κόστος εγγραφής του. Στο log-block FTL, όλες οι εγγραφές των σελίδων γίνονται στα log blocks, πριν την πραγματική τους εγγραφή στη μνήμη Flash, και στη συνέχεια τα log blocks συγχωνεύονται με τα data blocks. Όταν ένα log block εγγράφεται διαδοχικά από την πρώτη του σελίδα μέχρι την τελευταία, μπορεί απλώς να αντικαταστήσει το αντίστοιχο data block με μια λειτουργία switch merge. Αν ένα block θύμα που εγγράφεται από τον BPLRU είναι γεμάτο, τότε πραγματοποιείται μια λειτουργία switch merge στο log-block FTL. Διαφορετικά, πραγματοποιείται μια σχετικά δαπανηρή λειτουργία full merge αντί της λειτουργίας switch merge.

Επομένως, ο BPLRU διαβάζει μερικές σελίδες που δεν βρίσκονται στο block θύμα και εγγράφει όλες τις σελίδες του block διαδοχικά. Η τεχνική του page padding μπορεί να φαίνεται ότι πραγματοποιεί άσκοπες αναγνώσεις και εγγραφές, αλλά είναι πιο αποτελεσματική γιατί μπορεί να μετατρέψει μια δαπανηρή λειτουργία full merge σε μια αποτελεσματική λειτουργία switch merge.

Στην Εικόνα 5.12 παρουσιάζεται ένα παράδειγμα της τεχνικής page padding. Στο παράδειγμα αυτό, το block θύμα έχει μόνο δύο σελίδες (12 και 15), και ο BPLRU διαβάζει τις σελίδες 13 και 14 για το page padding. Τότε τέσσερις σελίδες (12 έως 16) εγγράφονται διαδοχικά. Στο log-block FTL, ένα log block χρησιμοποιείται για τις εγγραφές και αντικαθιστά το data block, εφόσον το log block εγγράφεται διαδοχικά για όλες τις σελίδες. Με άλλα λόγια πραγματοποιείται ένα switch merge.



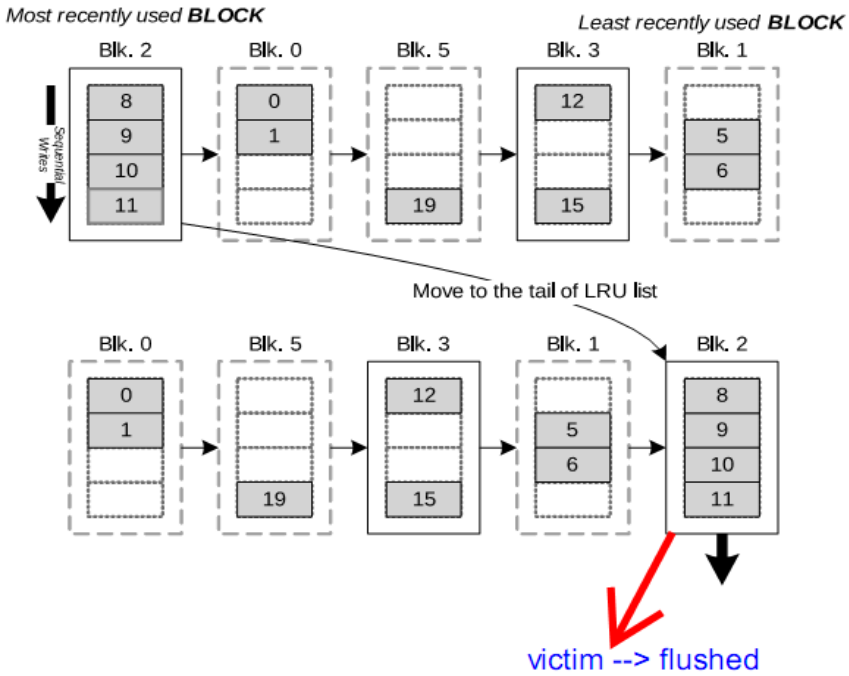
Εικόνα 5.12 : Page padding. Όταν ο Block-level LRU ένα block θύμα που περιέχει μόνο τις σελίδες 12 και 15, οι σελίδες 13 και 14 διαβάζονται από τη μνήμη Flash και στη συνέχεια εγγράφονται διαδοχικά και οι 4 σελίδες.

5.4.3 LRU Compensation

Επειδή η πολιτική LRU δεν είναι τόσο αποτελεσματική για διαδοχικές εγγραφές, έχουν προταθεί μερικοί ενισχυμένοι αλγόριθμοι, όπως για παράδειγμα ο low inter-reference recency set (LIRS) και ο adaptive replacement cache (ARC), οι οποίοι παρουσιάστηκαν σύντομα προηγουμένως.

Για την αποζημίωση του LRU όσον αφορά τις διαδοχικές εγγραφές, χρησιμοποιείται μια απλή τεχνική στον BPLRU. Αν το πιο πρόσφατα χρησιμοποιημένο block έχει γραφεί διαδοχικά, υπολογίζεται ότι το συγκεκριμένο block έχει τη μικρότερη πιθανότητα να επανεγγραφεί στο άμεσο μέλλον. Οπότε, το block αυτό μεταφέρεται στην ουρά της λίστας LRU. Το σχήμα αυτό είναι επίσης πολύ σημαντικό όταν εφαρμόζεται η τεχνική του page padding.

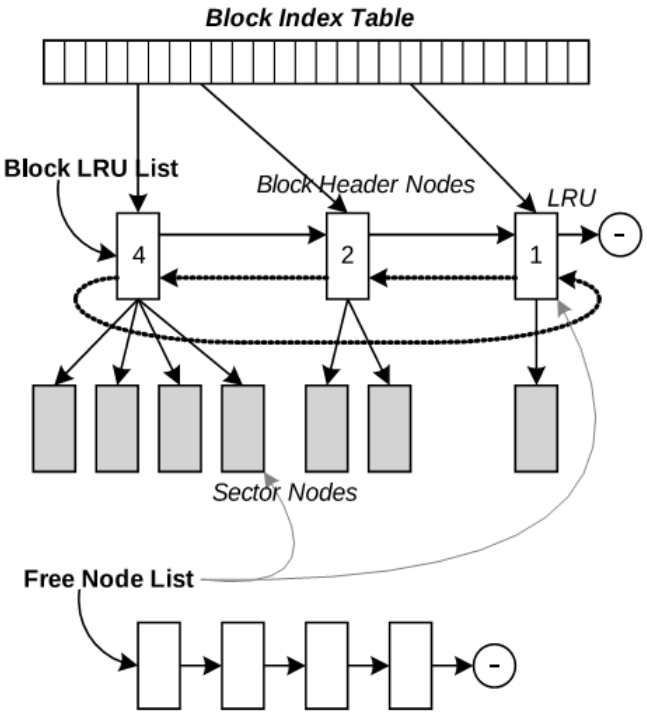
Η Εικόνα 5.13 εμφανίζει ένα παράδειγμα του LRU compensation. Ο BPLRU αναγνωρίζει ότι το block 2 έχει γραφεί διαδοχικά και το μεταφέρει στην πυρά της λίστας LRU. Όταν χρειαστεί περισσότερος ελεύθερος χώρος αργότερα, το συγκεκριμένο block πρόκειται να επιλεγεί ως θύμα και να εγγραφεί στη μνήμη Flash.



Εικόνα 5.13 : LRU Compensation. Όταν εγγράφεται η σελίδα 11, ο BPLRU αναγνωρίζει ότι το block 2 έχει γραφεί πλήρως διαδοχικά και το μεταφέρει στην ουρά της λίστας LRU.

Το πιο σημαντικό κομμάτι της υλοποίησης του LRU είναι η γρήγορη εύρεση της θέσης του buffer, καθώς η αναζήτηση αυτή απαιτείται για κάθε αίτηση ανάγνωσης και εγγραφής. Για τον σκοπό αυτό, οι σχεδιαστές του BPLRU χρησιμοποίησαν μια τεχνική ευρετηρίου (indexing) δύο επιπέδων. Στην Εικόνα 5.14 φαίνεται η δομή δεδομένων του BPLRU.

Με την τεχνική των δύο επιπέδων, μπορεί να βρεθεί ο κόμβος για τη σελίδα που ενδιαφερόμαστε χρησιμοποιώντας απλώς δύο index references. Για να βρεθεί ο κόμβος με τον ζητούμενο αριθμό σελίδας, πρώτα υπολογίζεται ο αριθμός block, διαιρώντας τον αριθμό σελίδας με τον αριθμό των σελίδων σε κάθε block, N. Αν δεν υπάρχει το συγκεκριμένο block, η σελίδα δεν βρίσκεται στον buffer εγγραφών. Αν όμως το block βρεθεί, τότε υπολογίζουμε το υπόλοιπο της διαίρεσης του αριθμού σελίδας με το N, ούτως ώστε να βρούμε τη ζητούμενη σελίδα.



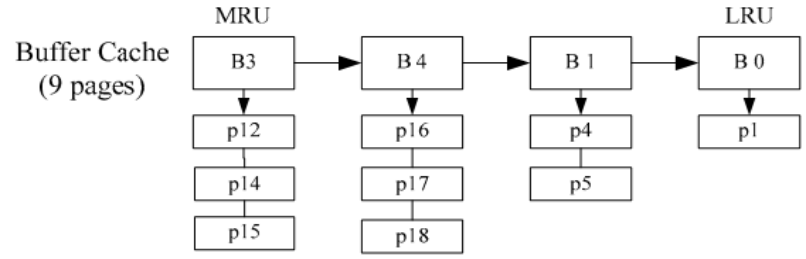
Εικόνα 5.14 : Η δομή δεδομένων του BPLRU

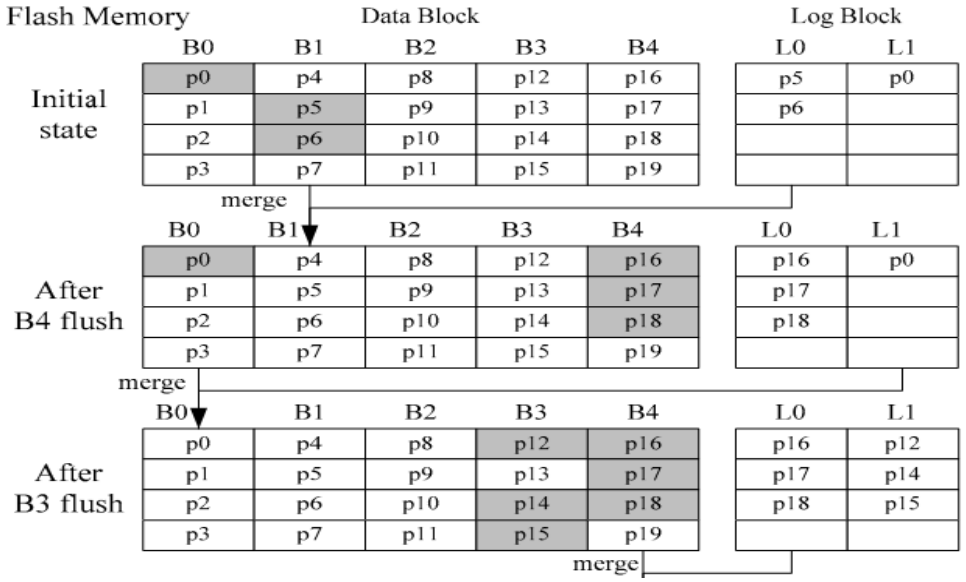
Το κύριο μειονέκτημα του BPLRU αφορά την τεχνική του Block-level LRU. Πιο συγκεκριμένα, αν μία μόνο σελίδα ενός block έχει προσπελαστεί πρόσφατα, τότε όλες οι υπόλοιπες σελίδες που ανήκουν στο ίδιο block παραμένουν στην buffer cache, ακόμα και αν δεν έχουν χρησιμοποιηθεί πρόσφατα. Επιπλέον, με την αναφορά σε μία μόνο σελίδα του block, όλες οι υπόλοιπες μεταφέρονται στην κεφαλή της λίστας του LRU, δηλαδή στη θέση MRU. Έτσι όμως, σπαταλάται πολύτιμος χώρος από τον buffer για τις μη πρόσφατα χρησιμοποιημένες σελίδες.

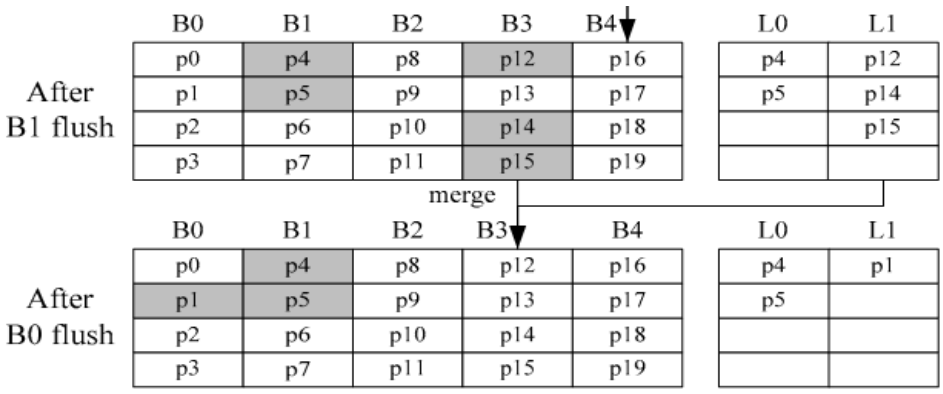
5.5 REF (Recently Evicted First) Buffer Replacement Policy for Flash Storage Devices

Τα προβλήματα του λυγισμού των log blocks και της υψηλής συσχέτισης των blocks (Ενότητα 3.2 Log-block FTL), μπορούν να μειωθούν χρησιμοποιώντας τεχνικές όπως ο FAB ή ο BPLRU. Αφού οι τεχνικές αυτές εγγράφουν όλες τις σελίδες ενός block θύματος στο log block, οι επακόλουθες εγγραφές σελίδων δεν προκαλούν αντικατάσταση του log block και δεν αυξάνουν το βαθμό συσχέτισης του log block.

Παρόλα αυτά, τα παραπάνω σχήματα δεν μπορούν να αποφύγουν τελείως το πρόβλημα του του λυγισμού των log blocks. Για παράδειγμα, στην Εικόνα 5.15 υπάρχουν 9 σελίδες στην buffer cache. Ο FAB διαχειρίζεται τη λίστα των blocks, η οποία είναι ταξινομημένη κατά recency, δηλαδή με βάση την τελευταία φορά που προσπελάστηκε μια σελίδα από ένα block. Κάθε κόμβος-block διαθέτει ένα δείκτη προς την αντίστοιχη λίστα των σελίδων που περιέχει. Όταν ο buffer γίνει πλήρης, ένας αριθμός σελίδων πρέπει να εκδιωχθεί. Στον FAB, το block που περιέχει τον μεγαλύτερο αριθμό σελίδων, επιλέγεται ως θύμα, το οποίο στο παράδειγμα είναι το block B4. Σε περίπτωση ισοπαλίας, επιλέγεται το λιγότερο χρησιμοποιημένο block. Αφού δύο log blocks, τα L0 και L1, έχουν ήδη αντιστοιχηθεί με τα data blocks B0 και B1, η συγχώνευση του log block πρέπει να γίνει σύμφωνα με το σχήμα BAST (1:1). Οι επακόλουθες εγγραφές των blocks B3, B1 και B0 προκαλούν επίσης συγχωνεύσεις όπως φαίνεται στο παρακάτω σχήμα.



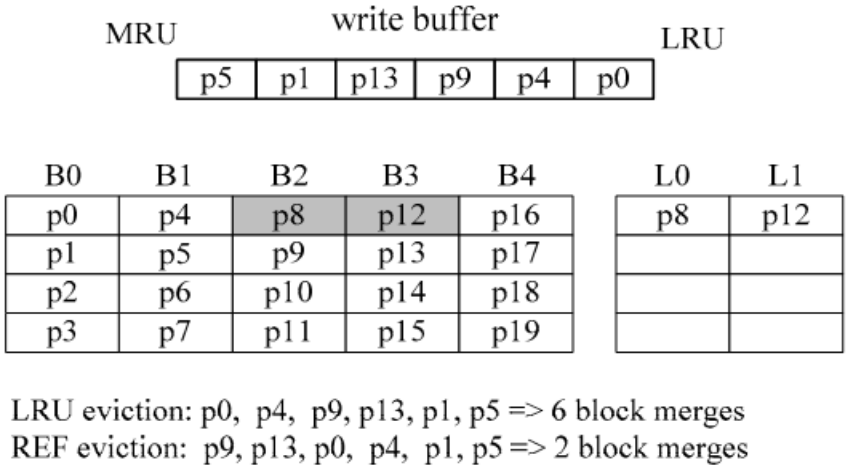




Εικόνα 5.15 : το πρόβλημα του λυγισμού των log blocks υπό τον FAB

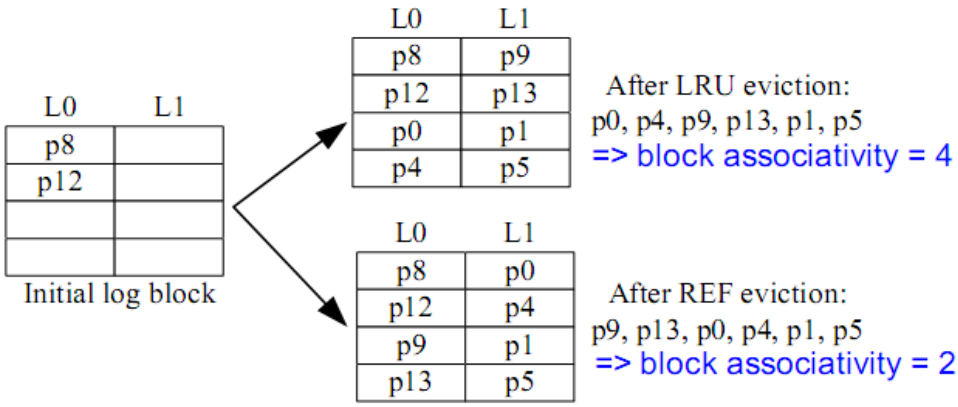
Τα προηγούμενα σχήματα διαχείρισης buffer δεν λαμβάνουν υπόψη τους τα log blocks στη μνήμη Flash. Αν η buffer cache επιλέξει μια σελίδα θύμα, τέτοια ώστε το αντίστοιχο data block της σελίδας να έχει συσχετιστεί με τα log blocks εκείνη τη στιγμή, τότε το πρόβλημα του λυγισμού των log blocks και το πρόβλημα της υψηλής συσχέτισης των log blocks μπορούν να αποφευχθούν.

Για να επιλέξει τα blocks θύματα που σχετίζονται με τα log blocks, ο REF τα προσδιορίζει χρησιμοποιώντας την πιο πρόσφατη εκδίωξη σελίδας. Για παράδειγμα, ας υποτεθεί ότι η buffer cache περιέχει τις σελίδες p0, p4, p9, p13, p1 και p5 ταξινομημένες σύμφωνα με τον LRU, δύο log blocks περιέχουν τις σελίδες p8 και p12, και τα log blocks διαχειρίζονται σύμφωνα με το σχήμα BAST, όπως φαίνεται στην Εικόνα 5.16. Αν χρησιμοποιηθεί ο αλγόριθμος αντικατάστασης LRU, οι σελίδες εκδιώκονται με τη σειρά p0, p4, p9, p13, p1, p5 και προκαλούνται 6 συγχωνεύσεις log block. Ωστόσο, αν αλλάξει η σειρά εκδίωξης των σελίδων, ο αριθμός των συγχωνεύσεων block μπορεί να μειωθεί. Αφού τα log blocks σχετίζονται με τα data blocks B2 και B3, είναι προτιμότερο να εκδιωχθούν πρώτα οι σελίδες p9 και p13. Έτσι, αν η εκδίωξη των σελίδων πραγματοποιηθεί με τη σειρά p9, p13, p0, p4, p1, p5, μόνο 2 συγχωνεύσεις log block απαιτούνται για το σχήμα BAST.



Εικόνα 5.16 : αντικατάσταση LRU vs αντικατάστασης REF για το σχήμα BAST

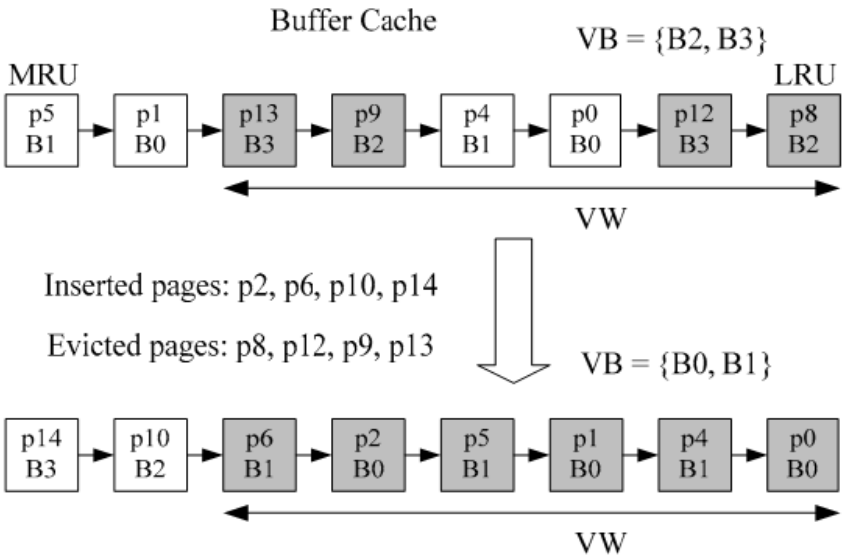
Για το σχήμα FAST, η συσχέτιση block για κάθε log block μπορεί να μειωθεί από τον REF. Η Εικόνα 5.17 παρουσιάζει τις αλλαγές των log blocks μετά την αντικατάσταση των σελίδων στην Εικόνα 5.16 αλλά για το σχήμα FAST. Αρχικά και η σελίδα p8 και η σελίδα p12 εγγράφονται στο log block L0, όταν χρησιμοποιείται το σχήμα FAST. Αν χρησιμοποιηθεί η πολιτική αντικατάστασης LRU, και τα δύο log blocks, L0 και L1, έχουν βαθμό συσχέτισης block ίσο με 4. Όμως, αν χρησιμοποιηθεί η πολιτική αντικατάστασης του REF, ο βαθμός συσχέτισης και για τα δύο log blocks μειώνεται στο 2. Επομένως, φαίνεται ότι ο REF μπορεί να μειώσει αποτελεσματικά και τον βαθμό συσχέτισης των log blocks, αλλά και τον αριθμό των συγχωνεύσεων.



Εικόνα 5.17 : αντικατάσταση LRU vs αντικατάστασης REF για το σχήμα FAST

Για να εκδιώξει τις σελίδες των οποίων τα αντίστοιχα data blocks σχετίζονται με τα log blocks, η πολιτική διαχείρισης του buffer πρέπει να γνωρίζει επακριβώς την κατάσταση των log blocks. Για να γίνει αυτό, το FTL θα πρέπει να παρέχει μια διεπαφή που να διερευνά την κατάσταση των log blocks. Παρόλα αυτά, ο REF μπορεί να υλοποιηθεί χωρίς να μεταβληθούν τα τρέχοντα FTL. Αντί να αναφέρεται άμεσα στην κατάσταση των log blocks, η πολιτική διαχείρισης του buffer επιλέγει μια σελίδα ως θύμα, χρησιμοποιώντας το πρόσφατο ιστορικό των εκδιώξεων από τον buffer.

Η Εικόνα 5.18 παρουσιάζει την αντικατάσταση των σελίδων υπό τον REF. Ας υποτεθεί ότι η buffer cache μπορεί να χωρέσει 8 σελίδες και οι σελίδες είναι ταξινομημένες σύμφωνα με τον LRU. Ο REF διατηρεί ένα σύνολο από blocks-θύματα (VB). Ο REF αναγκάζει τον buffer να εκδιώξει μόνο τις σελίδες που ανήκουν στα blocks-θύματα. Το πλήθος των blocks-θυμάτων πρέπει να είναι μικρότερο από το πλήθος των log blocks στη μνήμη Flash, για να αποτραπεί ο λυγισμός των log blocks. Στο παράδειγμα αυτό, το μέγεθος του VB είναι 2, (|VB| = 2). Ο REF επιλέγει τα υποψήφια blocks που μπορούν να μπουν στο VB χρησιμοποιώντας το παράθυρο των θυμάτων (VW), για να αποτρέψει την εκδίωξη των πρόσφατα χρησιμοποιημένων σελίδων. Για το συγκεκριμένο παράδειγμα, το μέγεθος του παραθύρου VW είναι 75%. Οπότε οι έξι λιγότερο χρησιμοποιημένες σελίδες (75% των 8 σελίδων) είναι υποψήφιες έξωση από τον buffer. Ο REF βρίσκει δύο blocks με τον μεγαλύτερο αριθμό σελίδων εντός του παραθύρου VW. Αρχικά, τα blocks B2 και B3 επιλέγονται ως θύματα. Στη συνέχεια, ο REF συνθέτει τη λίστα των σελίδων-θυμάτων με όλες τις σελίδες που βρίσκονται εντός του VW και των οποίων τα αντίστοιχα blocks ανήκουν στο VB.



Εικόνα 5.18 : επιλογή των block-θυμάτων υπό τον REF

Αν απαιτείται ελεύθερος χώρος στην buffer cache, εκδιώκεται η λιγότερο πρόσφατα χρησιμοποιημένη σελίδα. Αν υπάρχει σελίδα της οποίας το αντίστοιχο block ανήκει στο VB, αλλά η σελίδα δεν βρίσκεται εντός του παραθύρου VW, τότε η σελίδα μπορεί να μπει στο παράθυρο μετά την αντικατάσταση άλλων σελίδων. Τότε, η λίστα με τις σελίδες-θύματα ανανεώνεται εξαιτίας της εισαγωγής σε αυτή της νέας σελίδας.

Στην Εικόνα 5.18, μετά την εισαγωγή των σελίδων p2, p6, p10 και p14 στην buffer cache, οι σελίδες p8, p12, p9 και p13 εκδιώκονται διαδοχικά. Όταν δεν υπάρχει σελίδα στη λίστα των θυμάτων γιατί όλες οι σελίδες έχουν εγγραφεί στη μνήμη Flash, κατασκευάζεται ένα νέο VB. Στο παραπάνω σχήμα, μετά την εισαγωγή των σελίδων p2, p6, p10 και p14, τα blocks-θύματα γίνονται τα blocks B0 και B1.

Ενώ ο FAB και ο BPLRU αντικαθιστούν όλες τις σελίδες ενός block θύματος ταυτόχρονα, ο REF αντικαθιστά τις σελίδες μόνο κατά τον αριθμό των θέσεων που απαιτούνται. Η στρατηγική αυτή μπορεί να μειώσει το πλήθος των αποτυχιών στην cache, αν προσπελαστεί μια σελίδα-θύμα, και επομένως να μειώσει τον αριθμό των εγγραφών στη μνήμη Flash.

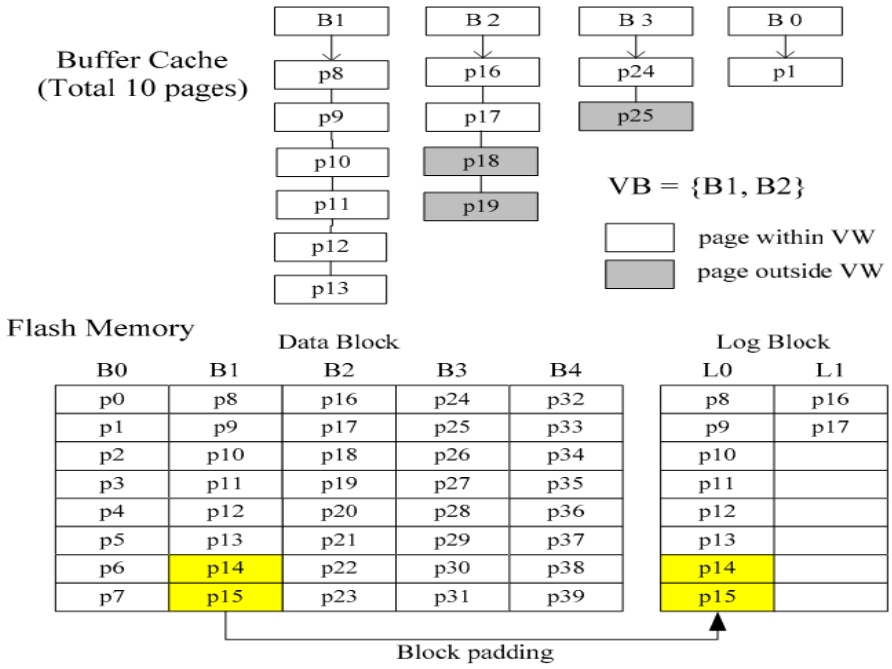
Το μέγεθος του παραθύρου VW πρέπει να επιλεγεί προσεκτικά, λαμβάνοντας υπόψη την τοπικότητα στο pattern των εγγραφών. Αν το μέγεθος του παραθύρου είναι πολύ μεγάλο, εκδιώκονται οι πρόσφατα χρησιμοποιημένες σελίδες, και ως αποτέλεσμα αυξάνεται το πλήθος των αποτυχιών στην buffer cache. Από την άλλη πλευρά, αν το μέγεθος του παραθύρου είναι πολύ μικρό, ο REF λειτουργεί όπως ο αυθεντικός LRU, και επομένως προκαλεί τον λυγισμό των log blocks. Σύμφωνα με τις πειραματικές μετρήσεις που πραγματοποίησαν οι σχεδιαστές του REF, το ιδανικό μέγεθος παραθύρου VW είναι το 75% της συνολικής χωρητικότητας του buffer.

Οι εισηγητές του REF πρότειναν επίσης τον BP-REF (Block Padding REF), ο οποίος χρησιμοποιεί επιπλέον την τεχνική του block padding, όπως ο BPLU, ο οποίος

εμφανίζει καλές επιδόσεις λόγω της τεχνικής αυτής. Ωστόσο, η τεχνική του block padding μπορεί να προκαλέσει ένα μεγάλο κόστος, αφού διαβάζει τις μη ενημερωμένες σελίδες από το αντίστοιχο data block, προκειμένου να εγγράψει ολόκληρο το block σε ένα log block της μνήμης Flash.
Επιπλέον, όταν χρησιμοποιείται η τεχνική του block padding, όλες οι σελίδες σε ένα log block εγγράφονται σε προκαθορισμένο σημείο μέσα στο block. Ως εκ τούτου, ο BPLRU μετατρέπει το log-block FTL σε block-level mapping FTL, με συνέπεια να κληρονομεί όλα τα μειονεκτήματα του συγκεκριμένου σχήματος. Ειδικά, στην περίπτωση που ενημερώνεται συχνά μόνο ένα μικρό μέρος ενός block (hot data), οι περισσότερες κρύες σελίδες στο block θα πρέπει να αντιγραφούν, μολονότι δεν έχουν μεταβληθεί. Όταν το μέγεθος του buffer είναι μικρό, το μέγεθος του του προβλήματος είναι κρίσιμο, καθώς ακόμα και οι ζεστές σελίδες δεν μπορούν να παραμείνουν για μεγάλο διάστημα στην buffer cache.

Για να αντισταθμίσει το μειονέκτημα του block padding, ο BP-REF χρησιμοποιεί την τεχνική αυτή επιλεκτικά. Μόνο όταν ο αριθμός των σελίδων-θυμάτων ενός block-θύματος είναι μεγαλύτερος από ένα κατώφλι, εφαρμόζεται η τεχνική του block padding. Για παράδειγμα, όταν το κατώφλι είναι 80% και ο buffer περιέχει περισσότερες σελίδες από το 80% του συνολικού αριθμού σελίδων του block, ο BP-REF διαβάζει το υπόλοιπο 20% των σελίδων του block από τη μνήμη Flash και εγγράφει ολόκληρο το block σε ένα log block. Η κατάλληλη τιμή για το κατώφλι του block padding εξαρτάται από το pattern πρόσβασης και το FTL που χρησιμοποιείται.

Η Εικόνα 5.19 παρουσιάζει τη συμπεριφορά του BP-REF. Τα επιλεγμένα blocks-θύματα είναι τα blocks B1 και B2. Όταν εκδιώκονται οι σελίδες του B1, ο BP-REF χρησιμοποιεί την τεχνική του block padding. Επομένως, διαβάζει τις σελίδες p14 και p15 και εγγράφει όλες τις σελίδες του block στο log block L0. Έτσι, αποτρέπει την εγγραφή των σελίδων του B2 στο log block L0, και ως αποτέλεσμα επιτρέπει στο FTL να πραγματοποιήσει μια λειτουργία switch merge.

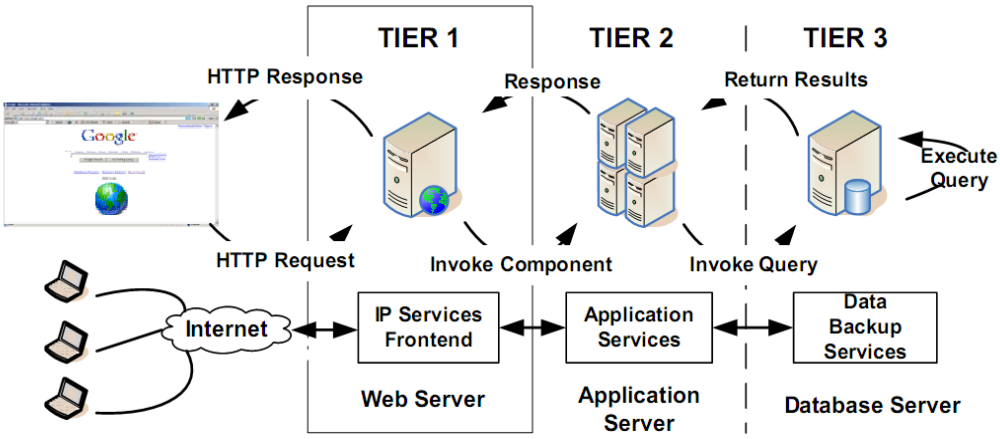


Εικόνα 5.19 : Block Padding στον REF

5.6 FlashCache : A NAND Flash Memory File Cache for Low Power Web Servers

Στην ενότητα αυτή παρουσιάζεται μια αρχιτεκτονική, η οποία δεν είναι χρησιμοποιείται ως αλγόριθμος αντικατάστασης buffer για τη μνήμη Flash, αλλά χρησιμοποιεί τη μνήμη Flash ως δευτερεύουσα buffer cache με σκοπό να μειώσει την κατανάλωση ενέργειας της κύριας μνήμης (RAM) στους web servers.

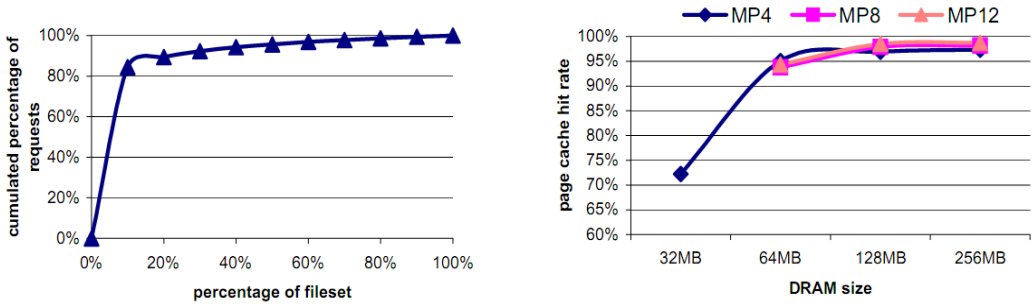
Όπως φαίνεται στην Εικόνα 7.1, οι web servers συνδέονται απευθείας στους clients και είναι υπεύθυνοι μόνο για την παράδοση του περιεχομένου στους clients.



Εικόνα 7.1 : μια τυπική αρχιτεκτονική τριών Tier servers. Tier1 – Web Server, Tier2 - Application Server, Tier3 – Database Server.

Αφού οι web servers απαιτούν απλώς μια μέτρια υπολογιστική ισχύ, η απόδοση τους εξαρτάται σε ένα πολύ μεγάλο βαθμό από την κύρια μνήμη, το εύρος ζώνης I/O και την καθυστέρηση προσπέλασης. Για να μετριάσουν την καθυστέρηση της Εισόδου/Εξόδου, του εύρους ζώνης αλλά κυρίως την καθυστέρηση προσπέλασης στους σκληρούς δίσκους, οι servers χρησιμοποιούν μεγάλες κύριες μνήμες, προσπαθώντας να αντιστοιχίσουν ολόκληρο το σύστημα αρχείων και να το αποθηκεύσουν στη μνήμη DRAM. Δυστυχώς, η DRAM καταναλώνει ένα πολύ μεγάλο μέρος από τη συνολική ισχύ του συστήματος. Οι σημερινοί τυπικοί servers διαθέτουν μεγάλες ποσότητες κύριας μνήμης, 4 ~ 32 GB, και καταναλώνουν περίπου 45 Watt σε κατάσταση αδράνειας (idle).

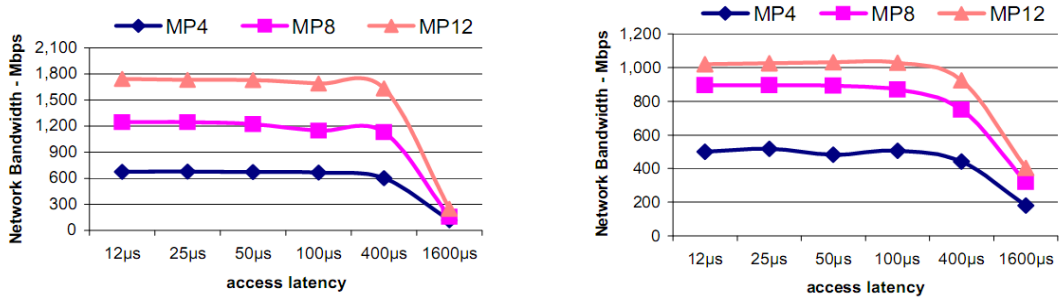
Η Εικόνα 7.2 δείχνει το ποσοστό επιτυχίας στην buffer cache για τις εφαρμογές των web servers. Η βελτίωση είναι οριακή από ένα σημείο και έπειτα. Ωστόσο, επειδή η καθυστέρηση προσπέλασης ενός σκληρού δίσκου είναι της τάξης των milliseconds, μια μεγάλη ποσότητα DRAM απαιτείται για να επανορθώσει για την καθυστέρηση του δίσκου. Διαφορετικά ο server δεν θα χρησιμοποιείται πλήρως, και θα παραμένει αδρανής καθώς περιμένει τα δεδομένα από το σκληρό δίσκο.



(a) (b)

Εικόνα 7.2 : (a). συνήθως το 90% των αιτήσεων αφορούν το 20% του περιεχομένου του server. (b). hit ratio σε σχέση με το μέγεθος του buffer

Από την Εικόνα 7.3 παρατηρείται ότι μια καθυστέρηση προσπέλασης έως και εκατοντάδες microseconds μπορεί να ανεχθεί χωρίς να υπάρχει κάποια αρνητική επίδραση στο throughput. Αυτό οφείλεται στην πολυνηματική φύση των εφαρμογών των web servers, η οποία επιτρέπει σε μια μέτρια καθυστέρηση της τάξης των microseconds να κρύβεται. Επιπλέον, οι αιτήσεις ανάγνωσης είναι πολύ συχνότερες από τις αιτήσεις εγγραφής για τις εφαρμογές των web servers, ενώ η κατανάλωση ενέργειας της μνήμης Flash είναι πολύ μικρότερη από εκείνη της DRAM. Επίσης, η μνήμη Flash είναι πιο φτηνή από τη DRAM, οπότε η μνήμη Flash θα μπορούσε να χρησιμοποιηθεί ως buffer cache για τη δευτερεύουσα μνήμη.



(a) SURGE (b) SPECWeb99

Εικόνα 7.3 : μέτρηση του bandwidth του δικτύου όταν μεταβάλεται η καθυστέρηση προσπέλασης στην δευτερεύουσα buffer cache.

	Density–Gb/cm^2	$/Gb	Active Power*	Idle Power*	Read Latency	Write Latency	Erase Latency	Built-in ECC support
DDR2 DRAM	0.7	48	878mW	80mW†	55ns	55ns	N/A	No
NOR	0.57	96	86mW	16μW	200ns	200μs	1.2s	No
NAND	1.42	21	27mW	6μW	25μs	200μs	1.5ms	Yes

* Power consumed for 1Gbit of memory

† DRAM Idle power in active mode. Idle power in powerdown mode is 18mW

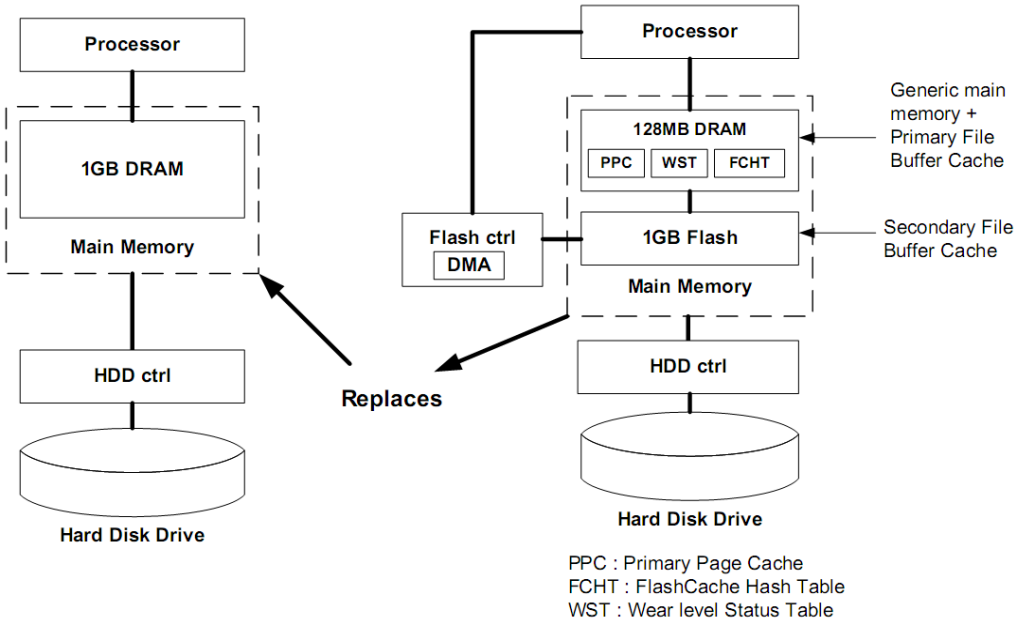
Πίνακας 7.4 : κόστος και κατανάλωση ενέργειας για συμβατικές DRAM, μνήμη Flash NOR και NAND

	2005	2007	2009	2011	2013	2016
Flash NAND Cell size –SLC/MLC*(μm^2)	0.0231/0.0116	0.0130/0.0065	0.0081/0.0041	0.0052/0.0013	0.0031/0.0008	0.0016/0.0004
Flash NOR Cell size(μm^2)†	0.0520	0.0293	0.0204	0.0117	0.0078	0.0040
DRAM Cell size(μm^2)	0.0514	0.0324	0.0153	0.0096	0.0061	0.0030
Flash erase/write cycles	1E+05	1E+05	1E+05	1E+06	1E+06	1E+07
Flash data retention	10-20	10-20	10-20	10-20	20	20

* SLC - Single level Cell, MLC - Multi Level Cell
† We assume a single level cell with smallest area size of $9F^2$ stated in the ITRS roadmap

Πίνακας 7.5 : η εξέλιξη της μνήμης Flash και της DRAM

Η Εικόνα 7.6 δείχνει την αρχιτεκτονική του συστήματος FlashCache. Σε σχέση με την αρχιτεκτονική ενός συμβατικού συστήματος που χρησιμοποιεί μόνο τη μνήμη DRAM, η αρχιτεκτονική FlashCache διαθέτει buffer cache δύο επιπέδων. Απαιτεί μια μικρή σχετικά ποσότητα κύριας μνήμης DRAM, η οποία χρησιμοποιείται ως η κύρια buffer cache, και μια μνήμη Flash που χρησιμοποιείται ως δευτερεύουσα buffer cache. Επιπλέον, απαιτείται ένας ελεγκτής για τη μνήμη Flash. Οι επιπρόσθετες δομές δεδομένων που χρειάζονται για τη διαχείριση της FlashCache τοποθετούνται στη μνήμη DRAM.



Εικόνα 7.6 : μια γενική άποψη της αρχιτεκτονικής FlashCache. Παρουσιάζεται ένα παράδειγμα όπου 1 GB DRAM αντικαθίσταται από 128 MB DRAM και 1 GB μνήμης Flash.

Οι δομές δεδομένων που χρησιμοποιούνται στην FlashCache διαβάζονται από τον σκληρό δίσκο και φορτώνονται στη μνήμη DRAM, για να μειωθεί η καθυστέρηση προσπέλασης και να μετριαστεί η φθορά των κελιών της μνήμης Flash.

Η καθυστέρηση προσπέλασης της DRAM είναι της τάξης των nanoseconds. Επειδή η αντίστοιχη καθυστέρηση της μνήμης Flash είναι μερικά microseconds, οι

σχεδιαστές της FlashCache χρησιμοποίησαν έναν πίνακα κατακερματισμού για να μειωθεί ο χρόνος αναζήτησης.

5.6.1 FlashCache Hash Table for tag lookup

Ο πίνακας κατακερματισμού της FlashCache (FCHT) είναι μια δομή που περιέχει ετικέτες (tags) σχετικά με τα blocks της μνήμης Flash. Ο πίνακας αυτός βρίσκεται στη μνήμη DRAM. Μια ετικέτα αποτελείται από το πεδίο διεύθυνσης σελίδας (page address field) και από το πεδίο διεύθυνσης μνήμης Flash (flash memory address field). Το πεδίο διεύθυνσης σελίδας δείχνει στην τοποθεσία στον σκληρό δίσκο και χρησιμοποιείται για να βρεθεί αν η μνήμη Flash περιέχει τη συγκεκριμένη θέση στον σκληρό δίσκο. Το αντίστοιχο πεδίο της διεύθυνσης μνήμης Flash χρησιμοποιείται για την πρόσβαση στη μνήμη Flash. Αν υπάρχει επιτυχία στον FCHT, το αντίστοιχο πεδίο της διεύθυνσης μνήμης Flash χρησιμοποιείται για την πρόσβαση στη μνήμη Flash. Για το 99% των αιτήσεων συμβαίνει επιτυχία γιο το σύστημα FlashCache και η τοποθεσία της μνήμης Flash που περιέχει το αιτούμενο αντικείμενο αποστέλλεται στην πρωτεύουσα buffer cache που υπάρχει στη DRAM. Αν ωστόσο υπάρξει αποτυχία, η FlashCache επιλέγει ένα block-θύμα, το οποίο εκδιώκεται από έναν αλγόριθμο αντικατάστασης που βασίζεται σε πληροφορίες σχετικές με τη φθορά των κελιών της μνήμης Flash.

5.6.2 Wear-level aware cache replacement

Ο πίνακας κατάστασης φθοράς (wear-level status table) που βρίσκεται στη DRAM διατηρεί τον αριθμό των εγγραφών και διαγραφών που έχουν πραγματοποιηθεί σε κάθε λογικό block. Ο αριθμός των εγγραφών και διαγραφών είναι ίσος με τον αριθμό των εκδιώξεων. Ένας μετρητής υπάρχει για κάθε λογικό block της μνήμης Flash και αυξάνεται κάθε φορά που επιλέγεται το συγκεκριμένο block για αντικατάσταση. Ο πίνακας κατάστασης φθοράς καθορίζει αν ένα block είναι ζεστό ή αν ένα set είναι ζεστό. Ένα set ή block είναι ζεστό αν ο αντίστοιχος μετρητής υπερβεί ένα κατώφλι.

Όταν υπάρχει αποτυχία στην FlashCache, σε επίπεδο block, επιλέγεται ένα block-θύμα για αντικατάσταση με βάση την πολιτική LRU. Ένα block είναι ζεστό, όταν η διαφορά του μετρητή του και του μικρότερου μετρητή ενός block που ανήκει στο ίδιο set, υπερβαίνει ένα κατώφλι. Αν το block-θύμα είναι ζεστό, τότε επιλέγεται ως θύμα το block με τον μικρότερο μετρητή για υπάρχει μια ισορροπία όσον αφορά τη φθορά των blocks.

Σε επίπεδο set, εξετάζεται αν το set είναι ζεστό ή όχι. Ένα set είναι ζεστό όταν η διαφορά του μεγαλύτερου μετρητή των blocks που περιέχει και του μετρητή άλλων set υπερβαίνει ένα κατώφλι. Ένα ζεστό set ανταλλάσσεται με ένα κρύο για να εξισορροπηθεί η φθορά.

5.6.3 Flash memory controller with DMA support

Ο ελεγκτής της μνήμης Flash υποστηρίζει το DMA για να χειρίζεται τις μεταφορές DMA από τη μνήμη DRAM στη μνήμη Flash και αντίστροφα. Η διαδικασία μεταφοράς δεδομένων από τη μνήμη Flash είναι απλή και μοιάζει με τη διαδικασία

προσπέλασης τη DRAM. Παρόλα αυτά, μπορεί να υπάρξουν δύο προβλήματα κατά την ανάγνωση ή εγγραφή της μνήμης Flash. Το πρώτο πρόβλημα αφορά το bandwidth. Συνήθως, η μνήμη Flash μπορεί να διαβάσει ή να εγγράψει ένα byte ή μία λέξη (word = 4 bytes) για κάθε κύκλο. Επιπρόσθετα, οι σημερινές μνήμες NAND Flash διαθέτουν ένα χαμηλό ρολόι – 50 Mhz. Επομένως, ένα μεγάλο ποσοστό του χρόνου χρησιμοποιείται για την ανάγνωση ή εγγραφή των δεδομένων. Αυτό μπορεί να είναι πρόβλημα όταν η μνήμη Flash προσπελάζεται συχνά. Το άλλο πιθανό πρόβλημα που μπορεί να προκύψει είναι το μπλοκάρισμα που προκαλούν οι εγγραφές. Οι σημερινές μνήμες Flash δεν υποστηρίζουν το Read While Write (RWW), δηλαδή όταν η μνήμη Flash είναι απασχολημένη από τις εγγραφές, μπλοκάρει όλες τις άλλες προσπελάσεις που μπορούν να προκύψουν.

Ευτυχώς, τα παραπάνω προβλήματα μπορούν να αντιμετωπισθούν. Όπως προαναφέρθηκε, είναι ανεκτή μια καθυστέρηση προσπέλασης εκατοντάδων microseconds, οπότε αυτό ανακουφίζει το πρόβλημα του περιορισμένου bandwidth της μνήμης Flash. Το πρόβλημα του μπλοκαρίσματος που προκαλούν οι εγγραφές, μπορεί να αντιμετωπιστεί με την κατανομή των εγγραφών. Πιο συγκεκριμένα, επειδή η συχνότητα των εγγραφών δεν είναι μεγάλη, μπορεί να δοθεί μεγαλύτερη προτεραιότητα στις αναγνώσεις χρησιμοποιώντας ένα σχήμα lazy write back. Με τη διαχείριση ενός write back buffer στη DRAM, οποίος αποθηκεύει τα blocks που πρέπει να ενημερωθούν στη μνήμη Flash, μια εγγραφή στη μνήμη Flash μπορεί να περιμένει όταν καταφθάνουν αιτήσεις ανάγνωσης.

5.6.4 Χειρισμός επιτυχιών και αποτυχιών στην FlashCache

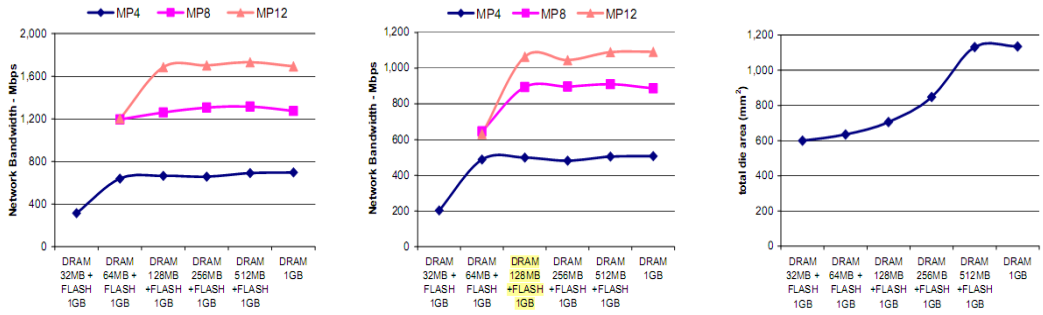
Αρχικά, το λειτουργικό σύστημα αναζητά το αιτούμενο αντικείμενο στην πρωτεύουσα buffer cache που βρίσκεται στη DRAM. Αν υπάρξει επιτυχία, η FlashCache δεν χρησιμοποιείται καθόλου, και το αντικείμενο διαβάζεται από την πρωτεύουσα buffer cache. Σε περίπτωση αποτυχίας, το λειτουργικό σύστημα ψάχνει στον πίνακα FCHT για να δει αν το αντικείμενο βρίσκεται στη δευτερεύουσα buffer cache. Αν βρεθεί, πραγματοποιείται μια λειτουργία ανάγνωσης της μνήμης Flash και προγραμματίζεται μια συναλλαγή DMA για τη μεταφορά του αντικειμένου στη DRAM. Η αιτούμενη διεύθυνση της μνήμης Flash αποκτάται από τον πίνακα FCHT.

Αν υπάρξει αποτυχία στον πίνακα FCHT, ένα λογικό block επιλέγεται για αντικατάσταση. Η διαδικασία επιλογής λαμβάνει υπόψη της αρχικά τη φθορά των blocks και στη συνέχεια τη φθορά των set, όπως προαναφέρθηκε στην Ενότητα 7.2. Ταυτόχρονα με την αντικατάσταση ενός λογικού block, προγραμματίζεται μια προσπέλαση στο σκληρό δίσκο, το περιεχόμενο του οποίου αντιγράφεται στην πρωτεύουσα buffer cache στη DRAM. Στη συνέχεια, ενημερώνεται η αντίστοιχη ετικέτα στον πίνακα FCHT. Τέλος, προγραμματίζεται μια εγγραφή (lazy write back) του περιεχομένου του σκληρού δίσκου στη μνήμη Flash χρησιμοποιώντας το DMA.

5.6.5 Αποτελέσματα της προσομοίωσης

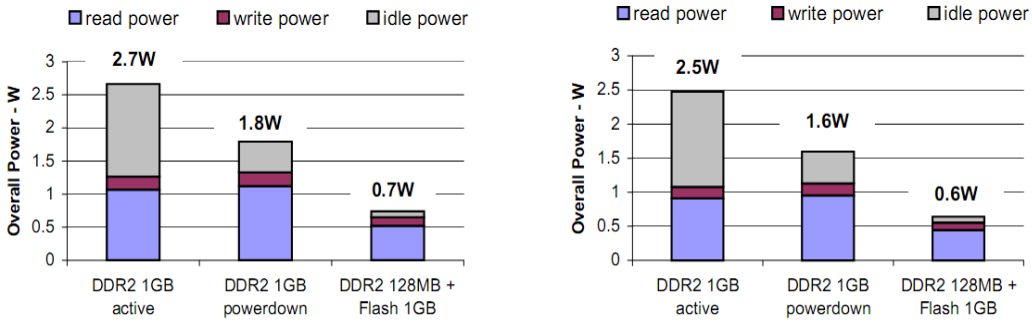
	Server configuration parameters
Processor type	single issue in-order
Number of cores	4, 8, 12 core
Clock frequency	1GHz
L1 cache size	4 way 16KB
L2 cache size	8 way 2MB
DRAM	64MB∼1GB t_{RC} latency 50ns bandwidth 6.4GB/s
NAND Flash Memory	1GB 16 way 128KB logical block size random read latency 25μs write latency 200μs erase latency 1.5ms bandwidth 50MB/s
IDE disk	average access latency 3.3ms bandwidth 300MB/s
Ethernet Device (NIC)	1Gbps Ethernet NIC
Number of NICs	2, 4, 6

Εικόνα 7.7 : διαμόρφωση του server



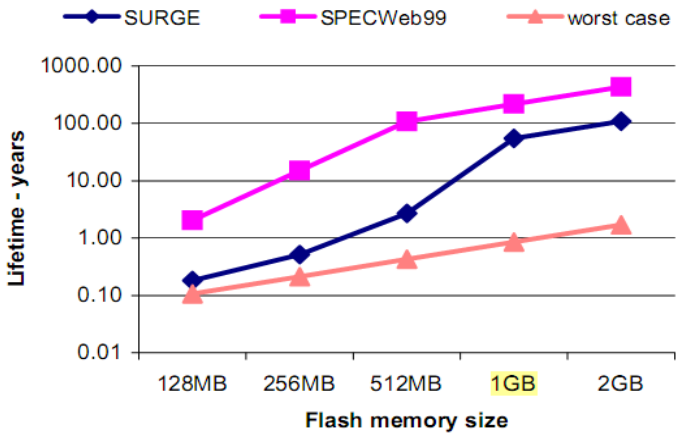
(a) SURGE (b) SPECWeb99 (c) total die area

Εικόνα 7.8 : το bandwidth του δικτύου σε σχέση με διάφορες ρυθμίσεις μνήμης



(a) SURGE (b) SPECWeb99

Εικόνα 7.9 : η συνολική κατανάλωση ενέργειας της μνήμης



Εικόνα 7.10 : προσδοκώμενος χρόνος ζωής της μνήμης Flash, εκτιμώντας την αντοχή της σε 1.000.000 κύκλους

6. LIRS-WSR

6.1 LIRS (Low Inter-reference Recency Set)

Όπως προαναφέρθηκε στην Ενότητα 4 (Παραδοσιακοί αλγόριθμοι αντικατάστασης Buffer), ο αλγόριθμος αντικατάστασης LIRS (low inter-reference recency set) είναι ένας ενισχυμένος αλγόριθμος, ο οποίος χρησιμοποιεί ως μέτρο σύγκρισης, εκτός από το πόσο πρόσφατα προσπελάστηκε μια σελίδα (recency), και το πόσο συχνά χρησιμοποιείται μια σελίδα (frequency). Ο LIRS διατηρεί δύο στοίβες μεταβλητού μεγέθους και διαχωρίζει τις σελίδες σε σελίδες LIR και σελίδες HIR. Οι σελίδες LIR είναι αυτές που έχουν προσπελαστεί ξανά, ενώ παραμένουν στην cache, ενώ οι σελίδες HIR είναι αυτές που δεν βρίσκονταν στην cache όταν προσπελάστηκαν.

Ο LIRS χρησιμοποιεί το IRR (Inter-Reference Recency) μιας σελίδας, το οποίο αναφέρεται στον αριθμό των διακριτών προσπελάσεων άλλων σελίδων, μεταξύ δύο συνεχόμενων αναφορών στη συγκεκριμένη σελίδα. Οι σελίδες LIR δεν επιλέγονται για αντικατάσταση, και δεν υπάρχουν σφάλματα σελίδας (page faults) για τις σελίδες αυτές. Μόνο ένα μικρό μέρος της cache χρησιμοποιείται για τις σελίδες HIR. Ουσιαστικά, ο LIRS επιλέγει ως θύμα τη σελίδα HIR με τη μεγαλύτερη τιμή recency ανάμεσα στις υπόλοιπες σελίδες HIR, δηλαδή τη λιγότερο πρόσφατα χρησιμοποιημένη σελίδα HIR. Όμως, όταν μια σελίδα LIR αποκτήσει μεγάλο recency και η τιμή του recency μιας σελίδας HIR ελαττωθεί και γίνει μικρότερη από την αντίστοιχη της σελίδας LIR, τότε οι σελίδες αυτές εναλλάσσονται, δηλαδή η σελίδα LIR γίνεται HIR, ενώ η HIR γίνεται LIR.

Μερικές από τις σελίδες HIR μπορεί να μη βρίσκονται στην cache, αλλά να παραμένουν εκεί τα metadata τους. Οι σελίδες αυτές ονομάζονται σελίδες HIR-non resident. Το μέγεθος της cache, έστω ότι είναι L σελίδες, χωρίζεται σε δύο κομμάτια, ένα πολύ μεγάλο και ένα πολύ μικρό. Το μεγάλο κομμάτι, του οποίου το μέγεθος έστω ότι είναι Llirs, χρησιμοποιείται για την αποθήκευση των σελίδων LIR, ενώ το μικρό κομμάτι της cache, του οποίου το μέγεθος συμβολίζεται με Lhirs, χρησιμοποιείται για την αποθήκευση των σελίδων HIR. Προφανώς, ισχύει Llirs + Lhirs = L. Όταν συμβεί σφάλμα στην cache και πρέπει να αντικατασταθεί μια σελίδα, επιλέγεται μια σελίδα HIR, η οποία είναι διαμένουσα στην cache, δηλαδή είναι σελίδα HIR-resident. Οι σελίδες LIR είναι πάντα διαμένουσες (resident) στην cache, οπότε δεν υπάρχουν σφάλματα για τις σελίδες αυτές. Παρόλα αυτά, μια προσπέλαση μιας σελίδας HIR είναι πολύ πιθανό να προκαλέσει σφάλμα, εξαιτίας του μικρού μεγέθους Lhirs. Πρακτικά, το Lhirs είναι το 1% του μεγέθους της cache, ενώ το Llirs είναι το 99%.

Στον Πίνακα 6.1, το σύμβολο “X” υποδηλώνει την προσπέλαση μιας σελίδας σε μια εικονική χρονική στιγμή. Για παράδειγμα, η σελίδα A προσπελάζεται τη χρονική στιγμή 1, 6 και 8. Την χρονική στιγμή 10, τα IRR των σελίδων A, B, C, D και E είναι 1, 1, άπειρο, 3 και άπειρο αντίστοιχα, ενώ οι τιμές του recency είναι 1, 3, 4, 2 και 0 αντίστοιχα. Αν υποτεθεί ότι η cache μπορεί να περιέχει 3 σελίδες κάθε χρονική στιγμή, Llirs = 2 και Lhirs = 1, ο αλγόριθμος LIRS διατηρεί δύο σελίδες LIR (A, B) και οι υπόλοιπες είναι σελίδες HIR (C, D, E).

Επειδή η σελίδα E είναι η πιο πρόσφατα χρησιμοποιημένη σελίδα, δηλαδή έχει τη μικρότερη τιμή recency, είναι η μοναδική HIR-resident σελίδα, καθώς ισχύει ότι Lhirs = 1. Οι υπόλοιπες σελίδες HIR είναι non resident. Αν προσπελαστεί μια σελίδα LIR, παραμένει στο σύνολο των σελίδων LIR. Αν όμως, προσπελαστεί μια σελίδα HIR, πρέπει να εξεταστεί το ενδεχόμενο η σελίδα αυτή να γίνει LIR.

Blocks / Virtual time	1	2	3	4	5	6	7	8	9	10	Recency	IRR
A	x					x		x			1	1
B			x		x						3	1
C				x							4	inf
D		x					x				2	3
E									x		0	inf

Πίνακας 6.1 : ένα παράδειγμα επιλογής σελίδας-θύματος από τον LIRS

Εφόσον, προσπελαστεί μια σελίδα HIR και το νέο IRR της γίνει μικρότερο από τη μεγαλύτερη τιμή του recency των σελίδων LIR, η σελίδα HIR γίνεται LIR, ενώ η σελίδα με τη μεγαλύτερη τιμή recency μετατρέπεται σε HIR. Με τον τρόπο αυτό, μια σελίδα HIR έχει τη δυνατότητα να προσχωρήσει στο σύνολο των σελίδων LIR, ενώ παράλληλα το μέγεθος του συνόλου LIR δεν γίνεται μεγαλύτερο από τη μέγιστη τιμή του, Llirs.

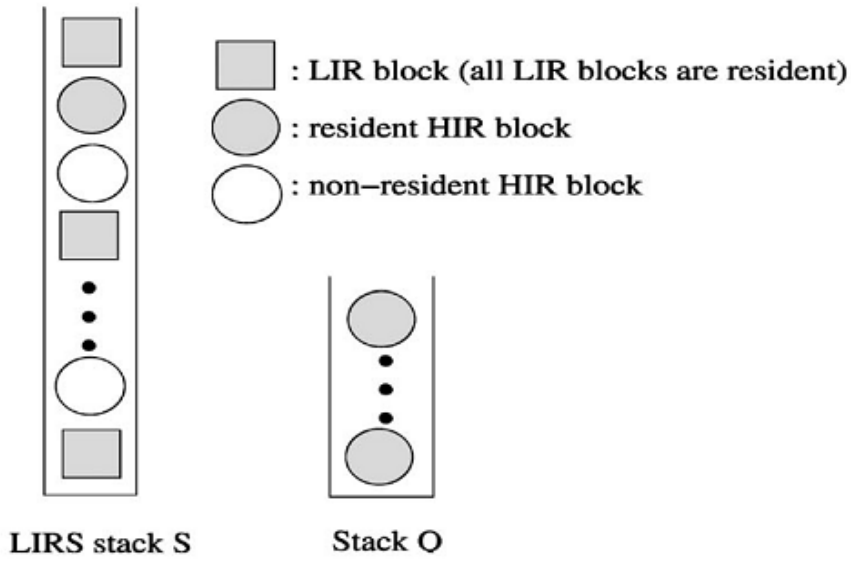
Στο παράδειγμα του πίνακα 6.1, αν προσπελαστεί η σελίδα D τη χρονική στιγμή 10, συμβαίνει ένα σφάλμα σελίδας (miss). Ο LIRS αντικαθιστά τη σελίδα HIR E, αντί της σελίδας B, η οποία θα είχε αντικατασταθεί από τον LRU εξαιτίας του γεγονότος ότι η σελίδα B έχει τη μεγαλύτερη τιμή recency μεταξύ των σελίδων A και E. Επιπλέον, το νέο IRR της σελίδας D είναι 2, το οποίο είναι μικρότερο από την τιμή recency της σελίδας LIR B (=3), επομένως η σελίδα D γίνεται LIR και η σελίδα B γίνεται HIR. Αφού η σελίδα B είναι πλέον η μοναδική σελίδα HIR-resident, πρόκειται να αφαιρεθεί από την cache μόλις χρειαστεί ελεύθερος χώρος. Αν τώρα τη χρονική στιγμή 10, προσπελαστεί η σελίδα C, της οποίας η τιμή recency είναι 4 ενώ της σελίδας D είναι 2, δεν πρόκειται να υπάρξει μετατροπή σελίδας HIR σε LIR και αντιστρόφως. Η σελίδα C θα γίνει HIR-resident, ενώ η σελίδα που θα αντικατασταθεί παραμένει η E. Με τον τρόπο αυτό, σχηματίζονται τα σύνολα των σελίδων LIR και HIR, ενώ παράλληλα μπορούν να μεταβληθούν δυναμικά κατά την πάροδο του χρόνου.

Ο αλγόριθμος LIRS μπορεί να υλοποιηθεί χρησιμοποιώντας τη στοίβα του LRU. Η στοίβα αυτή περιέχει L θέσεις μνήμης, κάθε μία από τις οποίες μπορεί να αποθηκεύσει μία σελίδα. Συνήθως, L είναι το μέγεθος της μνήμης cache. Σε αντίθεση με τη στοίβα του LRU, η οποία περιέχει μόνο διαμένουσες (resident) σελίδες, στον LIRS, αποθηκεύονται σελίδες LIR και HIR, των οποίων η τιμή recency δεν υπερβαίνει τη μέγιστη τιμή, σε μια στοίβα που ονομάζεται S. Η λειτουργία της στοίβας S είναι παρόμοια με στοίβα του LRU, μολονότι το μέγεθος της στοίβας S είναι μεταβλητό. Με τη σχεδίαση αυτή, δεν είναι απαραίτητη γνώση των τιμών IRR και recency για κάθε

σελίδα, αλλά και η αναζήτηση της μέγιστης τιμής recency μεταξύ των σελίδων που ανήκουν στο σύνολο LIR. Κάθε θέση στη στοίβα καταγράφει την κατάσταση LIR/HIR μιας σελίδας και το αν η σελίδα είναι resident ή non resident. Οι σελίδες HIR-resident διαμένουν σε μια μικρότερη στοίβα Q, με μέγεθος Lhirs. Όταν πρέπει να αντικατασταθεί μια σελίδα, ο LIRS εκδιώκει τη σελίδα HIR-resident που βρίσκεται στον πάτο της στοίβας Q. Ωστόσο, η σελίδα αυτή παραμένει στη στοίβα S και μετατρέπεται σε σελίδα HIR-non resident, αν βρισκόταν αρχικά και στη στοίβα S εκτός από τη στοίβα Q. Η σελίδα που βρίσκεται στον πάτο της στοίβας S πρέπει να είναι κάθε φορά μια σελίδα LIR, οπότε αφαιρούνται όλες οι σελίδες HIR που βρίσκονται κάτω από την τελευταία σελίδα LIR. Η διαδικασία αυτή ονομάζεται stack pruning. Όταν προσπελαστεί μια σελίδα HIR που βρίσκεται στη στοίβα S, το οποίο σημαίνει ότι υπάρχει τουλάχιστον μια σελίδα LIR, όπως για παράδειγμα η σελίδα στον πάτο της στοίβας S, της οποίας το νέο IRR θα είναι μεγαλύτερο από το αντίστοιχο της σελίδας HIR, η σελίδα HIR γίνεται LIR, ενώ η σελίδα LIR γίνεται HIR. Τότε, η σελίδα LIR που βρίσκεται στον πάτο της στοίβας S αφαιρείται και μπαίνει στην κορυφή της στοίβας Q ως σελίδα HIR-resident. Η σελίδα αυτή πρόκειται σύντομα να αντικατασταθεί, λόγω του μικρού μεγέθους της στοίβας Q, Lhirs.

Όταν η στοίβα S δεν είναι πλήρης, όλες οι σελίδες που εισέρχονται στην cache, μεταφέρονται στη στοίβα S και γίνονται σελίδες LIR, μέχρι να φτάσει το μέγεθος της στοίβας την οριακή τιμή Llirs. Από εκεί και έπειτα, κάθε νέα σελίδα που μπαίνει στην cache ή δεν βρίσκεται πλέον στην cache επειδή έχει να χρησιμοποιηθεί για αρκετό χρονικό διάστημα, γίνεται HIR.

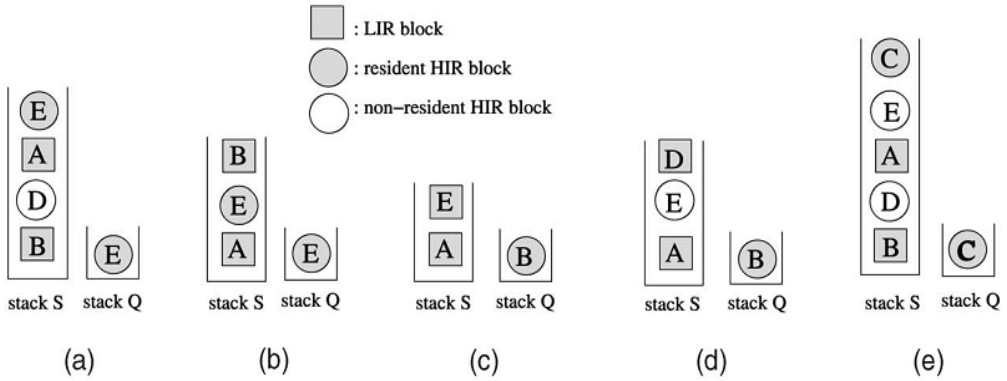
Η Εικόνα 6.2 παρουσιάζει μια περίπτωση όπου η στοίβα S περιέχει τρία είδη σελίδων : σελίδες LIR, σελίδες HIR-resident και σελίδες HIR-non resident. Η σελίδα Q περιέχει μόνο τις σελίδες HIR-resident. Μια σελίδα HIR μπορεί να βρίσκεται στη στοίβα S ως resident ή ως non-resident.



Εικόνα 6.2 : οι στοίβες S και Q του αλγορίθμου LIRS

Υπάρχουν τρεις περιπτώσεις για τις αναφορές σε αυτές τις σελίδες, οι οποίες εμφανίζονται στην Εικόνα 6.3 χρησιμοποιώντας το παράδειγμα του πίνακα 6.1.

1. Προσπέλαση μιας σελίδας LIR X. Η περίπτωση αυτή είναι σίγουρα μια επιτυχία στην cache. Η σελίδα X μεταφέρεται στην κορυφή της στοίβας S. Αν η σελίδα LIR βρισκόταν προηγουμένως στον πάτο της στοίβας S, εφαρμόζεται η διαδικασία του stack pruning. Η περίπτωση αυτή απεικονίζεται στη μεταφορά από την κατάσταση (a) στην κατάσταση (b) στην Εικόνα 6.3.
2. Προσπέλαση μιας σελίδας HIR-resident X. Και σε αυτή την περίπτωση υπάρχει επιτυχία στην cache. Η σελίδα X μεταφέρεται στην κορυφή της στοίβας S. Υπάρχουν δύο περιπτώσεις για την προηγούμενη θέση της σελίδας X : a). Αν η σελίδα X βρισκόταν προηγουμένως στη στοίβα S, η σελίδα γίνεται LIR και αφαιρείται από τη στοίβα Q. Η σελίδα LIR που βρίσκεται στον πάτο της στοίβας S μεταφέρεται στην κορυφή της στοίβας Q και γίνεται HIR. Στη συνέχεια, εφαρμόζεται η διαδικασία του stack pruning. Η περίπτωση αυτή απεικονίζεται στη μεταφορά από την κατάσταση (a) στην κατάσταση (c) στην Εικόνα 6.3.
   b). Αν η σελίδα X δεν βρισκόταν προηγουμένως στη στοίβα S, η σελίδα μεταφέρεται και στην κορυφή της στοίβας Q, εκτός από την S, ενώ παραμένει HIR.
3. Προσπέλαση μιας σελίδας HIR-non resident X. Στην περίπτωση αυτή υπάρχει αποτυχία στην cache. Αφαιρείται η σελίδα HIR που βρίσκεται στον πάτο της στοίβας Q και γίνεται non resident. Έπειτα, η σελίδα X μεταφέρεται στην κορυφή της στοίβας S. Υπάρχουν δύο περιπτώσεις για την προηγούμενη θέση της σελίδας X : a). Αν η σελίδα X βρισκόταν προηγουμένως στη στοίβα S, η σελίδα μετατρέπεται σε LIR και η σελίδα LIR που βρίσκεται στον πάτο της στοίβας S μεταφέρεται στην κορυφή της στοίβας Q και μετατρέπεται σε σελίδα HIR. Η περίπτωση αυτή απεικονίζεται στη μεταφορά από την κατάσταση (a) στην κατάσταση (d) στην Εικόνα 6.3.
   b). Αν η σελίδα X δεν βρισκόταν προηγουμένως στη στοίβα S, η σελίδα μεταφέρεται και στην κορυφή της στοίβας Q, εκτός από την S, ενώ παραμένει HIR. Η περίπτωση αυτή απεικονίζεται στη μεταφορά από την κατάσταση (a) στην κατάσταση (e) στην Εικόνα 6.3.



Εικόνα 6.3 : έστω ότι η περίπτωση (a) αντιστοιχεί στη χρονική στιγμή 9 του πίνακα 6.1. Οι προσπελάσεις των σελίδων B, E, D ή C τη χρονική στιγμή 10 απεικονίζονται στις μεταβάσεις (b), (c), (d) και (e) από την κατάσταση (a)

6.2 LIRS-WSR (Write Sequence Reordering)

Η πολιτική WSR (Write Sequence Reordering) και ο αλγόριθμος αντικατάστασης LIRS-WSR έχουν σχεδιαστεί για την buffer cache συστημάτων, των οποίων η δευτερεύουσα μνήμη είναι μνήμη Flash. Ο στόχος του αλγορίθμου LIRS-WSR είναι η μείωση του αριθμού των εγγραφών των βρόμικων σελίδων στη μνήμη Flash όταν συμβαίνει αντικατάστασης σελίδας. Για να επιτύχει αυτόν τον στόχο, χρησιμοποιεί την ακόλουθη στρατηγική : καθυστερεί όσο το δυνατό περισσότερο την εκδίωξη των σελίδων που είναι βρόμικες και έχουν μεγάλη συχνότητα πρόσβασης. Χρησιμοποιώντας τη στρατηγική αυτή, το ποσοστό επιτυχίας στην cache χρησιμοποιώντας τον LIRS-WSR μπορεί να είναι μικρότερο από εκείνο του LIRS, με αποτέλεσμα να αυξάνεται το πλήθος των αναγνώσεων από τη μνήμη Flash. Παρόλα αυτά, ο αλγόριθμος αυτός μειώνει σημαντικά τον αριθμό των εγγραφών των σελίδων στη μνήμη Flash, αλλά και το πλήθος των διαγραφών της. Ως αποτέλεσμα, αυξάνει τη συνολική απόδοση του συστήματος με δευτερεύουσα αποθήκευση τη μνήμη Flash.

6.2.1 WSR (Write Sequence Reordering)

Ο αλγόριθμος αντικατάστασης CFLRU (Ενότητα 5.1) διατηρεί τις βρόμικες σελίδες στον buffer, χωρίς να λαμβάνει υπόψη του τη συχνότητα προσπέλασης των σελίδων αυτών. Όπως προαναφέρθηκε, η λειτουργία αυτή μπορεί να ελαττώσει τη συνολική απόδοση του συστήματος, καθώς υποβαθμίζεται το ποσοστό επιτυχίας στην cache.

Για να αντιμετωπιστεί το παραπάνω πρόβλημα, χρησιμοποιείται η τεχνική του WSR. Το βασικό σχήμα του WSR είναι το εξής :

1). Η χρησιμοποίηση αλγορίθμου που να μπορεί να κρίνει αν μια σελίδα είναι κρύα ή ζεστή
2). Η καθυστέρηση εγγραφής των σελίδων που είναι βρόμικες και ζεστές.

Για την υλοποίηση του WSR, μόνο ένα πεδίο που ονομάζεται cold-flag απαιτείται, για να μπορεί να κρίνει ο αλγόριθμος αν μια σελίδα είναι κρύα ή ζεστή. Όταν η πολιτική αντικατάστασης επιλέγει μια σελίδα ως θύμα, εξετάζεται αν η σελίδα είναι βρόμικη. Αν η σελίδα είναι βρόμικη και δεν έχει ενεργοποιηθεί το πεδίο cold-flag, η σελίδα θεωρείται ως ζεστή βρόμικη σελίδα. Τότε, ενεργοποιείται το πεδίο cold-flag της σελίδας και η πολιτική αντικατάστασης προσπαθεί να επιλέξει μια άλλη σελίδα ως θύμα. Αν η υποψήφια σελίδα είναι καθαρή ή κρύα και βρόμικη, τότε επιλέγεται ως θύμα και εκδιώκεται από την cache. Επιπρόσθετα, το πεδίο cold-flag μιας βρόμικης σελίδας απενεργοποιείται όταν η σελίδα προσπελάζεται ξανά και επομένως θεωρείται ζεστή.

Ο WSR είναι ένας ευρετικός αλγόριθμος που βασίζεται στον αλγόριθμο δεύτερης ευκαιρίας. Είναι αποδεδειγμένο μέσω πειραμάτων ότι ο WSR μειώνει αποτελεσματικά τον αριθμό των εγγραφών και διαγραφών της μνήμης Flash, χωρίς να επιφέρει σημαντική υποβάθμιση του ποσοστού επιτυχίας στον buffer.

6.2.2 LIRS-WSR

Εφαρμόσαμε την πολιτική WSR στον αυθεντικό αλγόριθμο LIRS, προκειμένου να σχεδιάσουμε έναν ενισχυμένο LIRS, τον LIRS-WSR για την χρησιμοποίηση του στη μνήμη Flash. Οι διαφορές του αυθεντικού LIRS και του LIRS-WSR επισημαίνονται αμέσως μετά.

Στον LIRS-WSR, μόνο μια καθαρή σελίδα ή μια κρύα και ταυτόχρονα βρόμικη σελίδα μεταφέρεται από τον πάτο της στοίβας S στην κορυφή της στοίβας Q, σε αντίθεση με τον αυθεντικό LIRS, όπου η σελίδα που βρίσκεται στον πάτο της στοίβας S μεταφέρεται στην κορυφή της στοίβας Q, ανεξάρτητα αν είναι καθαρή ή βρόμικη, κρύα ή ζεστή.

Στον LIRS-WSR, μια ζεστή αλλά και βρόμικη σελίδα που βρίσκεται στον πάτο της στοίβας S μεταφέρεται στην κορυφή της ίδιας στοίβας, και στη συνέχεια ενεργοποιείται το πεδίο της, cold-flag. Δηλαδή, η σελίδα γίνεται κρύα.

Όταν προσπελάζεται μια σελίδα LIR, η οποία βρίσκεται (αναγκαστικά) στη στοίβα S, η σελίδα μεταφέρεται στην κορυφή της στοίβας S, και στη συνέχεια απενεργοποιείται το πεδίο της, cold-flag. Με άλλα λόγια, η σελίδα γίνεται ζεστή. Όταν προσπελάζεται μια σελίδα HIR, η οποία βρίσκεται στη στοίβα S, ο LIRS-WSR εξετάζει τη σελίδα (LIR) που βρίσκεται στον πάτο της στοίβας S. Αν η σελίδα στον πάτο της στοίβας S είναι καθαρή ή το πεδίο της, cold-flag, έχει ενεργοποιηθεί, δηλαδή η σελίδα είναι κρύα, η σελίδα αυτή μεταφέρεται στην κορυφή της στοίβας Q. Αν όμως η σελίδα είναι βρόμικη και το πεδίο της, cold-flag, δεν έχει ενεργοποιηθεί, δηλαδή η σελίδα είναι ζεστή, η συγκεκριμένη σελίδα μεταφέρεται στην κορυφή της στοίβας S, και στη συνέχεια γίνεται κρύα, δηλαδή ενεργοποιείται το πεδίο της, cold-flag. Έπειτα, ο LIRS-WSR εξετάζει την επόμενη σελίδα από τον πάτο της στοίβας S. Οι υπόλοιπες λειτουργίες του αλγορίθμου είναι ίδιες με τις αντίστοιχες του αυθεντικού LIRS.

6.3 Υλοποίηση του LIRS-WSR στη γλώσσα C

Υλοποιήσαμε τον LIRS-WSR στη γλώσσα C και τον συγκρίναμε με τους αλγορίθμους LIRS, LRU και OPTIMAL.

OPTIMAL είναι ο βέλτιστος αλγόριθμος, ο οποίος μεγιστοποιεί το ποσοστό επιτυχίας στην cache. Γενικά ο βέλτιστος αλγόριθμος δεν μπορεί να υλοποιηθεί, επειδή πρέπει να γνωρίζει πότε πρόκειται να προσπελαστεί κάθε σελίδα στο μέλλον. Ωστόσο, στην προσομοίωση αυτή, ο βέλτιστος αλγόριθμος μπορεί να γνωρίζει το μέλλον, διαβάζοντας τις αναφορές στις σελίδες από τα αρχεία traces, τα οποία αποτελούν τις εισόδους των αλγορίθμων. Η λειτουργία του βέλτιστου αλγόριθμου είναι η εξής : Όταν συμβεί σφάλμα σελίδας και ο buffer είναι πλήρης, ο βέλτιστος αλγόριθμος επιλέγει ως θύμα τη σελίδα με τη μεγαλύτερη προς τα εμπρός απόσταση (forward distance) από όσες διαμένουν στον buffer. Με άλλα λόγια, ο βέλτιστος αλγόριθμος επιλέγει τη σελίδα που πρόκειται να προσπελαστεί πιο μακριά στο μέλλον από όλες τις υπόλοιπες. Δηλαδή, ο αλγόριθμος μεταθέτει τις αποτυχίες στον buffer όσο το δυνατό μακρύτερα στο μέλλον. Ο βέλτιστος αλγόριθμος διατηρεί μια λίστα από σελίδες, η οποία είναι ταξινομημένη με βάση την προς τα εμπρός απόσταση κάθε σελίδας. Η κεφαλή της λίστας περιέχει τη σελίδα με τη μικρότερη προς τα εμπρός απόσταση, ενώ η ουρά τη σελίδα με τη μεγαλύτερη. Ως θύμα επιλέγεται πάντα η σελίδα που βρίσκεται στην ουρά της λίστας, δηλαδή η σελίδα που πρόκειται να χρησιμοποιηθεί αργότερα από όλες τις άλλες σελίδες που είναι αποθηκευμένες στον buffer.

Η υλοποίηση του αλγορίθμου LIRS-WSR αποτελείται από τα αρχεία lirs-wsr.h και lirs-wsr.c

lirs-wsr.h

```
/* parakatw xrisimopoioume tin orologia block, kathws o authentikos algorithmos LIRS sxediastike gia systimata, */
/* twn opoiwn i deutereuousa mnimi einai enas skliros diskos. Opote, me ton oro block, ennooume ta blocks tou   */
/* sklirou diskou i tis selides tis mnimis Flash.                                                                */


#include "stdio.h"
#include "string.h"
#include "stdlib.h"


/* to pososto tou buffer pou katalamvanoun oi selides HIR */
#define HIR_RATE 1.0


/* i metavliti auti kathorizei tin eikoniki xroniki stigmi, opou arxizoume na syllegoume statistika stoixeia */
#define STAT_START_POINT 0


#define LOWEST_HG_NUM 2                 // to elaxisto megethos tis stoivas Q (HIR)


#define TRUE 1
#define FALSE 0


#define S_STACK_IN 1                    // an i selida vrisketai sti stoiva S
#define S_STACK_OUT 0
```

```
typedef struct pf_struct {
  unsigned long ref_times;              // poses fores prospelastike i selida
  unsigned long pf_times;               // poses fores prokalese page fault, diladi poses fores DEN itan sti mnimi otan xreiastike

  unsigned long  page_num;              // o arithmos tis selidas
  int isResident;                    // 1 an i selida vrisketai sti mnimi
  int isHIR_block;                   // 1 an i selida einai HIR

  int isCold;                        // 1 an i selida einai Cold,  0 an i selida einai Hot
  int isClean;                       // 1 an i selida einai Clean, 0 an i selida einai Dirty

  struct pf_struct * LIRS_next;         // deiktis pros tin epomeni selida (eite LIR eite HIR) sti stoiva S
  struct pf_struct * LIRS_prev;         //     >>        proigoumeni      >>

  struct pf_struct * HIR_rsd_next;      // deiktis pros tin epomeni selida HIR (resident) sti stoiva Q
  struct pf_struct * HIR_rsd_prev;      //     >>        proigoumeni      >>

  unsigned int    recency;              // poso prosfata prospelastike i selida (monadiko metro sygkrisis gia ton LRU...)
} page_struct;                       // to page_struct antiproswpeuei mia selida

page_struct * page_tbl;                 // to * page_tbl einai enas pinakas apo selides


unsigned long total_pg_refs, warm_pg_refs; // o synolikos arithmos selidwn pou prospelastikan,
                                  // to idio alla apo to simeio pou theloume na arxisoun oi metriseis kai epeita

unsigned long num_pg_flt;               // o synolikos arithmos page faults, diladi misses sti mnimi

unsigned long num_physical_reads;       // to plithos twn fysikwn anagnwsewn apo ti mnimi flash
unsigned long num_physical_writes;      // to plithos twn fysikwn eggrafwn sti mnimi flash
float runtime;                       // o synolikos xronos ektelesis


long free_mem_size, mem_size, vm_size;    // eleutheri mnimi, synoliki mnimi, to megethos ris eikonikis mnimis

struct pf_struct * LRU_list_head;       // i koryfi tis stoivas S
struct pf_struct * LRU_list_tail;       // o patos    >>

struct pf_struct * HIR_list_head;       // i koryfi tis stoivas Q
struct pf_struct * HIR_list_tail;       // o patos    >>

struct pf_struct * LIR_LRU_block_ptr;     // i selida(LIR) pou vrisketai ston pato tis stoiva S

struct pf_struct * victim_block;        // to block-thyma pou metaferetai apo ton pato tiw stoivas S stin koryfi tis stoivas Q

struct pf_struct * temp_ptr;

unsigned long HIR_block_portion_limit;    // to megethos tis stoivas Q (hir) se selides

unsigned long cur_lir_S_len;            // to megethos tis stoivas S (kai me LIR kai me HIR selides)


/* synartiseis */
extern page_struct *find_last_LIR_LRU();
extern void add_HIR_list_head(page_struct * new_rsd_HIR_ptr);
extern void add_LRU_list_head(page_struct *new_ref_ptr);
extern FILE *openReadFile();
extern void prune_LIRS_stack();
extern page_struct *find_victim();
extern unsigned long calc_dirty_pages();
extern unsigned long calc_runtime();



/* vriskoume to euros twn selidwn pou prospelazontai, diladi ti selida me ton mikrotero arithmo kai ti selida me ton       */
/* megalytero arithmo, diladi to megethos tis eikonikis mnimis. Episis vriskoume to plithos twn selidwn pou prospelazontai  */
/* sto arxeio twn traces                                                                                                    */
int get_range(FILE *trc_fp, long *p_vm_size, long *p_trc_len)       // diavazei ta dedomena apo to arxeio .trc kai epistrefei
{                                                 // to vm_size kai to total_ref_pg
  long ref_blk;
```

```
 long count = 0;
 long min, max;
 char access[10];

 fseek(trc_fp, 0, SEEK_SET);            // metavainoume stin arxi tou arxeiou .trc

 fscanf(trc_fp, "%ld", &ref_blk);       // diavazoume ti selida pou prospelazetai apo to arxeio .trc
 max = min = ref_blk;
 fscanf(trc_fp, "%s", access);          // diavazoume ton typo tis prosvasis

 while (!feof(trc_fp)){                 // diavazoume mia selida ti fora, mexri na teleiwsei to arxeio
  if (ref_blk < 0)
    return FALSE;
  count++;                              // auksanoume to plithos twn selidwn pou diavastikan
  if (ref_blk > max)
    max = ref_blk;
  if (ref_blk < min)
    min = ref_blk;
  fscanf(trc_fp, "%ld", &ref_blk);
  fscanf(trc_fp, "%s", access);
 }

 printf(" [%d  %d] for %lu refs in the trace\n", min, max, count);
 fseek(trc_fp, 0, SEEK_SET);
 *p_vm_size = max;          // vm_size --> o megalyteros arithmos selidas pou diavazetai, diladi to megethos ris eikonikis mnimis
 *p_trc_len = count;
 return TRUE;
}
```

lirs-wsr.c

```
/* Eisodos :
 *          arxeio traces (.trc)
 *          arxeio parametrwn (.par), diladi to megethos tou buffer
 *
 * Eksodos :
 *          arxeio hit rate (.cuv): to pososto epityxias ston buffer se sxesi me to megethos tou
 *
 */

/*
 *  xrisi apo ti grammi entolwn : lirs-wsr ABC opou ABC einai to onoma tou arxeiou traces
 *  Den xreiazetai na grapsoume tin kataliksi tou arxeiou, diladi .trc gia to arxeio traces
 *  i .par gia to arxeio twn parametrwn. Ta 2 auta arxeia prepei na exoun to idio onoma, ABC.
 *
 */


#include "lirs-wsr.h"


main(int argc, char* argv[])
{
 FILE *trace_fp, *cuv_fp, *para_fp, *wr_fp, *run_fp;
 unsigned long i;
 char trc_file_name[100];
 char para_file_name[100];
 char cuv_file_name[100];
 char wr_file_name[100];
 char run_file_name[100];

 if (argc != 2){
  printf("%s file_name_prefix[.trc] \n", argv[0]);
  system("PAUSE");
  return;
 }
```

```
strcpy(para_file_name, argv[1]);
strcat(para_file_name, ".par");
para_fp = openReadFile(para_file_name);          // i openReadFile() vrisketai meta ti main() kai anoigei to arxeio .par
                                                 // gia anagnwsi
strcpy(trc_file_name, argv[1]);
strcat(trc_file_name, ".trc");
trace_fp = openReadFile(trc_file_name);

strcpy(cuv_file_name, argv[1]);
strcpy(wr_file_name, argv[1]);
strcpy(run_file_name, argv[1]);

strcat(cuv_file_name, "_LIRS-WSR.cuv");
strcat(wr_file_name, "_LIRS-WSR_writes.txt");
strcat(run_file_name, "_LIRS-WSR_runtime.txt");

cuv_fp = fopen(cuv_file_name, "w");              // anoigoume to arxeio .cuv gia EGGRAFI
wr_fp = fopen(wr_file_name, "w");
run_fp = fopen(run_file_name, "w");

if (!get_range(trace_fp, &vm_size, &total_pg_refs)){  // vriskei ton synoliko arithmo selidwn pou diavazontai sto arxeio .trc
  printf("trace error!\n");                      // kai to euros twn selidwn pou diavazontai, diladi ti selida me ton
  return;                                        // mikrotero arithmo kai ti selida me ton megalytero arithmo.
}


fscanf(para_fp, "%ld", &mem_size);               // diavazoume to trexwn megethos tis mnimis/buffer

page_tbl = (page_struct *)calloc(vm_size+1, sizeof(page_struct));   // desmeuoume mnimi

while (!feof(para_fp)){                          // diavazoume kathe fora to megethos tis mnimis apo to arxeio .par
  if (mem_size < 10){
    printf("WARNING: Too small cache size(%d). \n", mem_size);
    break;                                       // eksodos apo tin while()
  }

  printf("\nmem_size = %d\n", mem_size);

  total_pg_refs = 0;
  warm_pg_refs = 0;
  num_pg_flt = 0;

  num_physical_reads = 0;                        // to plithos twn fysikwn anagnwsewn apo ti mnimi flash
  num_physical_writes = 0;                       // to plithos twn fysikwn eggrafwn sti mnimi flash
  runtime = 0;                                   // o synolikos xronos ektelesis

  cur_lir_S_len = 0;                             // to megethos tis stoivas S (kai me LIR kai me HIR selides)

  fseek(trace_fp, 0, SEEK_SET);                  // metakinoumaste stin arxi tou arxeiou .trc

  free_mem_size = mem_size;

  /* arxikopoioume tis selides */
  for (i = 0; i <= vm_size; i++){
    page_tbl[i].ref_times = 0;
    page_tbl[i].pf_times = 0;

    page_tbl[i].page_num = i;
    page_tbl[i].isResident = 0;
    page_tbl[i].isHIR_block = 1;                 // arxika oles oi selides einai HIR

    page_tbl[i].isCold = 1;                      // arxika oles oi selides einai Cold
    page_tbl[i].isClean = 1;                     //    >>       >>          Clean

    page_tbl[i].LIRS_next = NULL;
    page_tbl[i].LIRS_prev = NULL;

    page_tbl[i].HIR_rsd_next = NULL;
    page_tbl[i].HIR_rsd_prev = NULL;
```

```
        page_tbl[i].recency = S_STACK_OUT;              // giati den vriskontai sti stoiva S. Otan mpoun ekei, tote recency = S_STACK_IN
    }

    LRU_list_head = NULL;                       // anaferetai sti stoiva S. Oi selides mporei na einai eite LIR eite HIR
    LRU_list_tail = NULL;

    HIR_list_head = NULL;
    HIR_list_tail = NULL;

    LIR_LRU_block_ptr = NULL;                   // i ligotero prosfati selida LIR (vrisketai ston pato tis stoivas S)

    victim_block = NULL;

    /* to pososto tis synolikis mnimis pou katalamvanoun oi selides HIR einai 1% */
    /* to megethos tis stoivas S (lirs) einai to 99% tis cache                   */
    HIR_block_portion_limit = (unsigned long)(HIR_RATE/100.0*mem_size);   // HIR_RATE --> 1.0
     if (HIR_block_portion_limit < LOWEST_HG_NUM)                         // LOWEST_HG_NUM --> 2, opote i stoiva Q exei megethos
      HIR_block_portion_limit = LOWEST_HG_NUM;                            // to ligotero 2 selides

    printf(" Lhirs (cache size for HIR blocks) = %d\n", HIR_block_portion_limit);

    LIRS_WSR_Repl(trace_fp);                // i LIRS_WSR_Repl() vrisketai parakatw kai kanei oli ti douleia ...

    printf("total blocks refs = %d  number of misses = %d \nhit rate = %2.1f\%, mem shortage ratio = %2.3f\% \n", total_pg_refs,
                                        num_pg_flt, (1-(float)num_pg_flt/warm_pg_refs)*100, (float)mem_size/vm_size*100);

       fprintf(cuv_fp,  "%5d   %2.1f\n",  mem_size,  100-(float)num_pg_flt/warm_pg_refs*100);   // grafoume sto arxeio .cuv to megethos
                                                    // tis mnimis kai to pososto twn hits
    fprintf(wr_fp, "%5d  %lu \n", mem_size, num_physical_writes);

    fprintf(run_fp, "%5d  %.1f \n", mem_size, runtime);

    fscanf(para_fp, "%ld", &mem_size);              // diavazoume to epomeno megethos mnimis apo to arxeio .par
  } // telos tis while()

  fflush(NULL);
  printf("\n");
  printf("Press <ENTER> to exit the simulation ...\n");
  getchar();
  return;
} // telos tis main()



FILE *openReadFile(char file_name[])
{
  FILE *fp;

  fp = fopen(file_name, "r");          // anoigoume to arxeio me onoma file_name gia anagnwsi

  if (!fp) {
    printf("can not find file %s.\n", file_name);
    return NULL;
  }

  return fp;
}



LIRS_WSR_Repl(FILE *trace_fp)
{
  unsigned long ref_block, i, j, step;
  long num_LIR_pgs = 0;
  int collect_stat = (STAT_START_POINT==0)?1:0;       // STAT_START_POINT --> 0 (default),  opote   collect_stat = 1
  int count=0;
  char access[10];                          // o typos tis prosvasis (read/write)
```

```
int read_count=0, write_count=0;                    // plithos anagnwsewn, plithos eggrafwn
float write_percentage=0;                           // to pososto twn eggrafwn


fseek(trace_fp, 0, SEEK_SET);                       // metavainoume stin arxi tou arxeiou

fscanf(trace_fp, "%lu", &ref_block);                // diavazoume ton arithmo tis selidas pou prospelazetai apo to arxeio .trc
fscanf(trace_fp, "%s", access);                     // diavazoume ton typo tis prosvasis, R = read, W = write request

i = 0;
while (!feof(trace_fp)){                            // synexizoume wsotou diavasoume olokliro to arxeio .trc
 total_pg_refs++;

 if( strcmp(access, "R") == 0 )                     // an o typos tis prosvasis einai anagnwsi, auksanoume ton antistoixo metriti
    read_count++;
 if( strcmp(access, "W") == 0 )                     //      >>          >>      eggrafi,      >>          >>
    write_count++;

 if (total_pg_refs % 10000 == 0)
  fprintf(stderr, "%d samples processed\r", total_pg_refs);
 if (total_pg_refs > STAT_START_POINT){
  collect_stat = 1;
  warm_pg_refs++;                                   // o synolikos arithmos selidwn pou prospelastikan alla apo to simeio
 }                                                  // pou theloume na arxisoun oi metriseis kai epeita

 if (ref_block > vm_size){
  printf("Wrong ref page number found: %d.\n", ref_block);
  return FALSE;
 }


 if (!page_tbl[ref_block].isHIR_block)              // an i selida einai (idi) LIR
  page_tbl[ref_block].isCold = 0;                   // (arxika oles oi selides einai HIR), tote ginetai Hot

 if( strcmp(access, "W") == 0 )                     // an o typos tis prosvasis einai eggrafi,
  page_tbl[ref_block].isClean = 0;                  // i selida ginetai Dirty



/* miss, page fault */
 if (!page_tbl[ref_block].isResident) {             // an exoume miss, diladi i selida einai HIR non-resident
  if (collect_stat == 1)
            num_pg_flt++;                           // auksanoume ton metriti twn page faults

  num_physical_reads++;                             // auksanoume ton metriti twn fysikwn anagnwsewn apo ti mnimi flash

  if (free_mem_size == 0) {                         // an kai oi 2 stoives (S, Q) einai gemates ...
    //        |   | top (end)                       // i stoiva tou LRU einai to idio pragma me ti lista
    //        |   |                                 // i koryfi tis stoivas antistoixei sto telos tis listas
    //        |   |                                 // kai o patos tis stoivas antistoixei stin arxi
    //        |__| bottom (front) --> VICTIM
    //       stack
            HIR_list_tail->isResident = FALSE;

    if (HIR_list_tail->isClean == 0)                // an i selida ston pato tis stoivas Q einai Dirty
      num_physical_writes++;                        // auksanoume ton metriti twn fysikwn eggrafwn sti mnimi flash

    HIR_list_tail->isCold = 1;                      // i selida ginetai cold kai clean se periptwsi pou ksanampei
    HIR_list_tail->isClean = 1;                     // argotera ston buffer
            remove_HIR_list(HIR_list_tail);         // afairoume ti selida pou vrisketai ston pato tis stoivas Q
            free_mem_size++;
  }
  else if (free_mem_size > HIR_block_portion_limit){    // arxika oles oi selides mpainoun sti stoiva S ws LIR
    page_tbl[ref_block].isHIR_block = FALSE;        // molis i stoiva S gemisei (den isxyei i anisotita),
            num_LIR_pgs++;                          // oi nees selides einai HIR
  }
  free_mem_size--;                                  // meiwnoume tis eleutheres theseis mnimis kata 1,
                                                    // afou i selida den vrisketai ston buffer
}
/* epityxia (hit) stin cache */
```

```
  else if (page_tbl[ref_block].isHIR_block)                    // an exoume hit kai i selida einai hir (resident stin Q)
    remove_HIR_list((page_struct *)&page_tbl[ref_block]);          // afairoume ti selida apo ti stoiva Q



  // o kwdikas apo edw kai katw ekteleitai eite exoume hit eite exoume miss ...

  remove_LIRS_list((page_struct *)&page_tbl[ref_block]);  // afairoume proswrina ti selida gia na ti metaferoume stin koryfi
                                        // tis stoivas S. An den vrisketai stin S den ginetai kammia leitourgia

  add_LRU_list_head((page_struct *)&page_tbl[ref_block]);          // vazoume ti selida pou molis diavastike stin koryfi
  page_tbl[ref_block].isResident = TRUE;                    // tis stoivas S (eite einai LIR eite einai HIR)
  if (page_tbl[ref_block].recency == S_STACK_OUT)              // an i selida den vriskotan sti stoiva S, auksanoume
    cur_lir_S_len++;                                    // to megethos tis stoivas S (kai me LIR kai me HIR selides)
                                                // an ypirxe apo prin tote den xreiazetai na to auksisoume

   if (page_tbl[ref_block].isHIR_block && (page_tbl[ref_block].recency == S_STACK_IN)){ // an einai HIR kai vriskotan sti
stoiva S
    page_tbl[ref_block].isHIR_block = FALSE;                          // eite einai resident, eite den einai
    num_LIR_pgs++;                                          // i selida metatrepetai apo HIR se LIR

    if (num_LIR_pgs > mem_size-HIR_block_portion_limit){// an i stoiva S einai yperpliris (mem_size + 1)
      victim_block = find_victim();                 // vriskoume ti selida thyma pou tha metaferthei sti lista Q symfwna me WSR
            add_HIR_list_head(victim_block);              // i selida thyma pou vrikame proigoumenws ginetai HIR kai
            HIR_list_head->isHIR_block = TRUE;            // metaferetai stin koryfi tis stoivas Q
            HIR_list_head->recency = S_STACK_OUT;         // to HIR_list_head einai pleon i selida thyma
            num_LIR_pgs--;                              // meiwnetai o arithmos twn selidwn LIR
            LIR_LRU_block_ptr = find_last_LIR_LRU();      // vriskoume tin epomeni selida LIR apo ton pato tis stoivas S
    }
    else
      printf("Warning2!\n");
  }
  else if (page_tbl[ref_block].isHIR_block)                    // an einai HIR kai DEN vrisketai sti stoiva S
    add_HIR_list_head((page_struct *)&page_tbl[ref_block]);       // i selida mpainei kai stin koryfi tis stoivas Q

  page_tbl[ref_block].recency = S_STACK_IN;           // molis mpike sti stoiva S, i ypirxe apo prin ...

  prune_LIRS_stack();                           // kladeuoume ti stoiva S, diladi afairoume oles tis selides HIR
                                        // apo ton pato tis stoivas S mexri na katsei ston pato mia selida LIR

  fflush(NULL);
  fscanf(trace_fp, "%lu", &ref_block);              // diavazoume tin epomeni selida
  fscanf(trace_fp, "%s", access);                   // kai tin epomeni prosvasi
  fflush(NULL);
 } // telos tis while (!feof(trace_fp))

 write_percentage = (float)write_count/total_pg_refs*100;
 printf("read_count : %d \t write_count : %d \t write_percentage = %2.1f \%\n", read_count, write_count, write_percentage);
 printf("num_physical_reads : %lu \t\t num_physical_writes : %lu \n", num_physical_reads, num_physical_writes);

 runtime = calc_runtime();
 printf("runtime = %.1f milliseconds \n", runtime);

 return;
} // telos tis LIRS_WSR_Repl()



/* afairei mia selida apo ti stoiva S (eite exoume hit eite exoume miss) */
int remove_LIRS_list(page_struct *page_ptr)
{
 if (!page_ptr)
   return FALSE;
                                        // arxika LIRS_prev kai LIRS_next einai NULL, diladi
 if (!page_ptr->LIRS_prev && !page_ptr->LIRS_next)          // an i selida den vriskotan sti stoiva S
   return TRUE;                                 // den xreiazetai na tin afairesoume apo ti stoiva

 if (page_ptr == LIR_LRU_block_ptr){                    // an i selida (LIR) vrisketai ston pato tis stoivas S
   LIR_LRU_block_ptr = page_ptr->LIRS_prev;
   LIR_LRU_block_ptr = find_last_LIR_LRU();             // vriskoume tin prwti LIR selida apo ton pato tis stoivas S (LRU)
```

```
  }

  if (!page_ptr->LIRS_prev)
    LRU_list_head = page_ptr->LIRS_next;
  else
    page_ptr->LIRS_prev->LIRS_next = page_ptr->LIRS_next;

  if (!page_ptr->LIRS_next)
    LRU_list_tail = page_ptr->LIRS_prev;
  else
    page_ptr->LIRS_next->LIRS_prev = page_ptr->LIRS_prev;

  page_ptr->LIRS_prev = page_ptr->LIRS_next = NULL;
  return TRUE;
}



/* afairoume mia selida (HIR) apo ti stoiva Q */
int remove_HIR_list(page_struct *HIR_block_ptr)       // kaleitai kai otan diagrafoume mia selida apo ti stoiva Q,
{                                                     // alla kai otan exoume hit stin cache kai i selida metaferetai
  if (!HIR_block_ptr)                                 // apo ti stoiva Q sti stoiva S
    return FALSE;

  if (!HIR_block_ptr->HIR_rsd_prev)
    HIR_list_head = HIR_block_ptr->HIR_rsd_next;
  else
    HIR_block_ptr->HIR_rsd_prev->HIR_rsd_next = HIR_block_ptr->HIR_rsd_next;

  if (!HIR_block_ptr->HIR_rsd_next)
    HIR_list_tail = HIR_block_ptr->HIR_rsd_prev;
  else
    HIR_block_ptr->HIR_rsd_next->HIR_rsd_prev = HIR_block_ptr->HIR_rsd_prev;

  HIR_block_ptr->HIR_rsd_prev = HIR_block_ptr->HIR_rsd_next = NULL;

  return TRUE;
}



/* vriskoume ti teleutaia selida LIR apo ton pato tis stoivas S */
page_struct *find_last_LIR_LRU()
{

  if (!LIR_LRU_block_ptr){
    printf("Warning*\n");
    exit(1);
  }

  while (LIR_LRU_block_ptr->isHIR_block == TRUE){          // oso i teleutaia selida sti stoiva S einai HIR
    LIR_LRU_block_ptr->recency = S_STACK_OUT;
    cur_lir_S_len--;                                       // meiwnoume kata 1 thesi to megethos tis stoivas S
    LIR_LRU_block_ptr = LIR_LRU_block_ptr->LIRS_prev;      // kai eksetazoume tin epomeni selida apo ton pato tis stoivas S
  }                                                        // mexri na vroume kapoia selida LIR

  return LIR_LRU_block_ptr;
}



/* kladeuoume ti stoiva S, diladi afairoume oles tis selides HIR apo ton pato tis stoivas */
void prune_LIRS_stack()
{
  page_struct * tmp_ptr;

  tmp_ptr = LRU_list_tail;

  while (tmp_ptr->isHIR_block)  {
      tmp_ptr->LIRS_prev->LIRS_next = tmp_ptr->LIRS_next;
```

```
        LRU_list_tail = tmp_ptr->LIRS_prev;

        tmp_ptr->LIRS_prev = tmp_ptr->LIRS_next = NULL;
        if (tmp_ptr->isClean == 0)                    // an i selida einai Dirty, auksanoume
            num_physical_writes++;                    // ton metriti twn fysikwn eggrafwn sti mnimi flash
        tmp_ptr->isCold = 1;                          // i selida ginetai cold kai clean se periptwsi pou ksanampei
        tmp_ptr->isClean = 1;                         // argotera ston buffer

        tmp_ptr = LRU_list_tail;
    }

    return;
}



/* vazoume mia selida stin koryfi tis  stoivas Q */
void add_HIR_list_head(page_struct * new_rsd_HIR_ptr)
{
    new_rsd_HIR_ptr->HIR_rsd_next = HIR_list_head;
    if (!HIR_list_head)
        HIR_list_tail = HIR_list_head = new_rsd_HIR_ptr;
    else
        HIR_list_head->HIR_rsd_prev = new_rsd_HIR_ptr;

    HIR_list_head = new_rsd_HIR_ptr;

    return;
}



/* vazoume mia selida (eite einai LIR eite einai HIR) stin koryfi tis stoivas S */
void add_LRU_list_head(page_struct *new_ref_ptr)
{
    new_ref_ptr->LIRS_next = LRU_list_head;

    if (!LRU_list_head){                          // an i stoiva S einai adeia
        LRU_list_head = LRU_list_tail = new_ref_ptr;          // i selida ginetai kai head kai tail,
        LIR_LRU_block_ptr = LRU_list_tail;        // alla kai LIR_LRU_block_ptr
    }
    else {
        LRU_list_head->LIRS_prev = new_ref_ptr;
        LRU_list_head = new_ref_ptr;
    }

    return;
}



/* vriskoume ti selida-thyma pou tha metaferthei apo ton pato tis stoivas S stin koryfi tis stoivas Q */
page_struct *find_victim()
{
    victim_block = LIR_LRU_block_ptr;

    if (!victim_block){
        printf("Warning : NO LIR_LRU_block_ptr !!! \n");
        exit(1);
    }

    // an i selida ston pato tis stoivas S einai HIR i einai LIR kai dirty kai hot, eksetazoume tin amesws epomeni selida apo ton
    // pato tis stoivas S.
    while ( victim_block->isHIR_block  ||  !(victim_block->isClean || victim_block->isCold) )  {
        if ( !(victim_block->isHIR_block) )    // an i selida einai LIR, dirty kai hot, tote ginetai Cold
            victim_block->isCold = 1;

        // eite einai HIR eite einai LIR, dirty kai hot, afairoume ti selida apo ton pato kai tin metaferoume stin koryfi tis stoivas S
        LIR_LRU_block_ptr = victim_block->LIRS_prev;
```

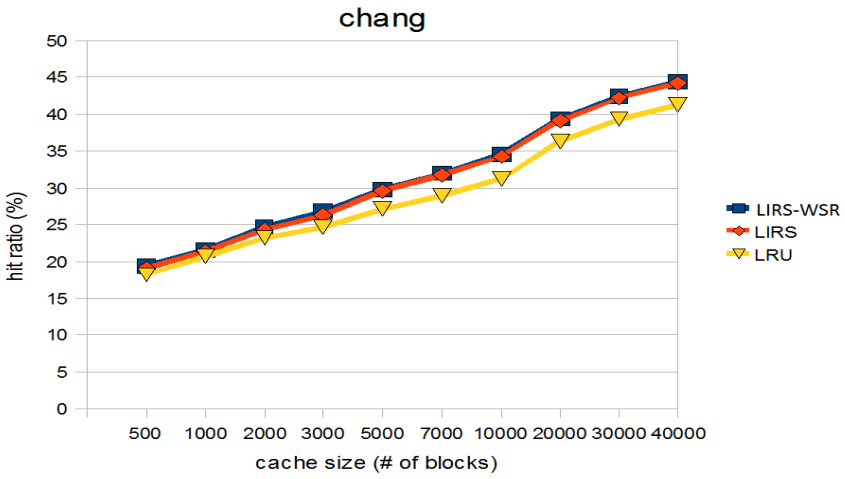
```
    if (!victim_block->LIRS_prev)
      LRU_list_head = victim_block->LIRS_next;
    else
      victim_block->LIRS_prev->LIRS_next = victim_block->LIRS_next;

    if (!victim_block->LIRS_next)
      LRU_list_tail = victim_block->LIRS_prev;
    else
      victim_block->LIRS_next->LIRS_prev = victim_block->LIRS_prev;

    victim_block->LIRS_prev = victim_block->LIRS_next = NULL;

    add_LRU_list_head(victim_block);

    victim_block = LIR_LRU_block_ptr;
   } // telos tis while()

   return victim_block;
}



/* ypologizoume to synoliko xrono ektelesis */
unsigned long calc_runtime()
{
  float read_latency=0.025;      // se ms
  float write_latency=0.25;      // se ms
  unsigned long runtime=0;

  runtime = (num_physical_reads * read_latency) + (num_physical_writes * write_latency);

  return runtime;
}
```

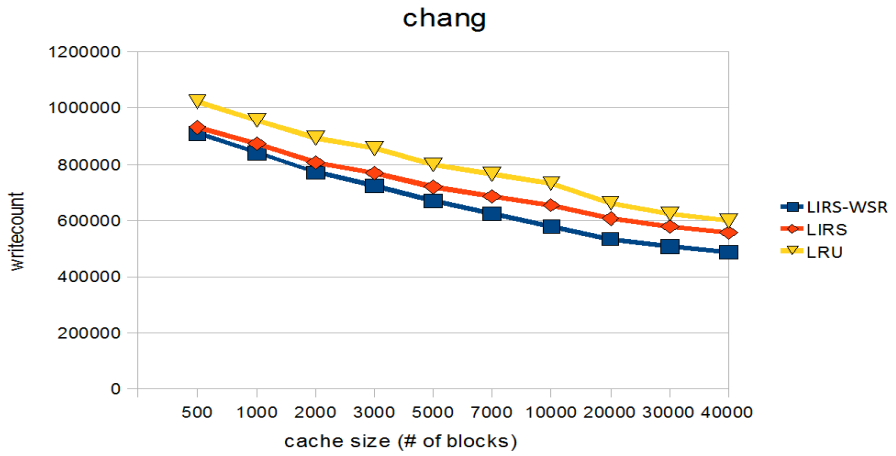
6.4 Αποτελέσματα της προσομοίωσης

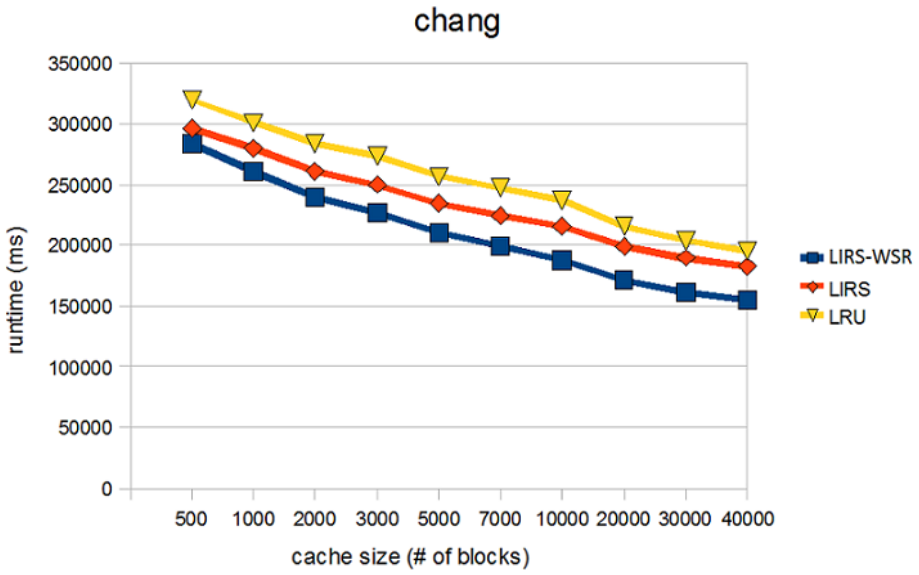
Στο σημείο αυτό συγκρίνουμε τα αποτελέσματα της προσομοίωσης για τους αλγόριθμους LIRS-WSR, LIRS, LRU και OPTIMAL. Πιο συγκεκριμένα παρουσιάζουμε για κάθε αρχείο traces το ποσοστό των επιτυχιών στην cache, το πλήθος των εγγραφών στη μνήμη Flash και τον συνολικό χρόνο εκτέλεσης.

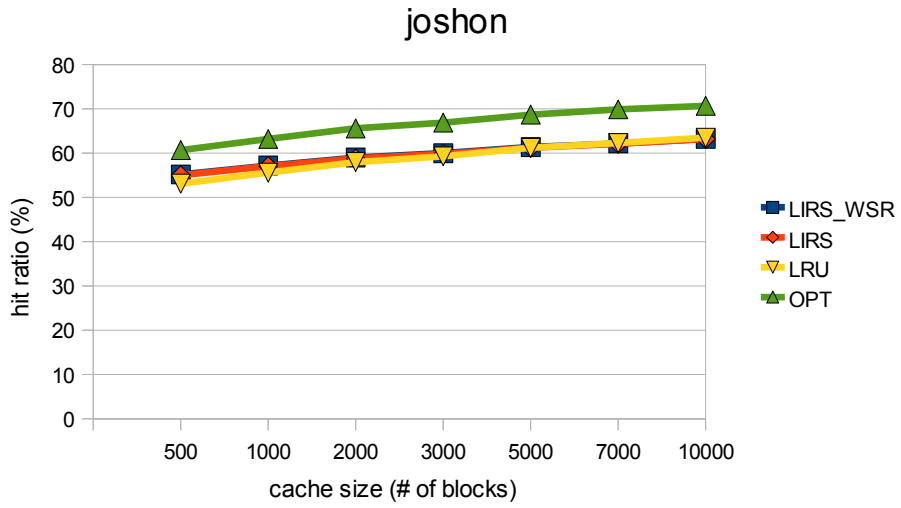
Το κύριο μειονέκτημα του LIRS-WSR, είναι ότι όταν η τοπικότητα στις εγγραφές (write locality) είναι χαμηλή, η πολιτική WSR μπορεί να μην είναι αποτελεσματική, και να μειώσει τη συνολική απόδοση. Αυτό οφείλεται στο γεγονός ότι το όφελος της μείωσης του αριθμού των εγγραφών στη μνήμη Flash γίνεται μικρότερο από το επιπλέον κόστος που παράγεται από την αύξηση του πλήθους των αναγνώσεων, εξαιτίας της υποβάθμισης του ποσοστού επιτυχίας στην buffer cache.

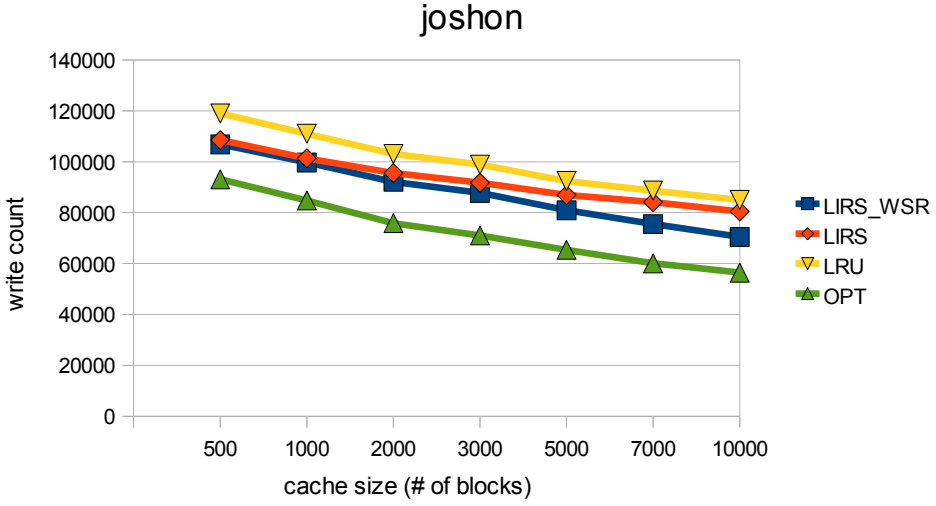
Χρησιμοποιήσαμε τα αρχεία traces των Dr. Li-Pin Chang (Dr. Li-Pin Chang@ the NEWS Lab of NTU), Joshon (Joshon @ the NEWS Lab of NTU) και Po-Liang (Po-Liang @ the NEWS Lab of NTU), τα οποία συλλέχθηκαν σε περίοδο ενός μήνα εκτελώντας εφαρμογές όπως περιήγηση στο web, λήψη/αποστολή e-mail, αναπαραγωγή ταινίας, εγγραφή κειμένου και εκτέλεση ηλεκτρονικών παιχνιδιών. Το πρώτο αρχείο traces δημιουργήθηκε σε σύστημα αρχείων FAT32, ενώ τα υπόλοιπα δύο σε σύστημα αρχείων NTFS.

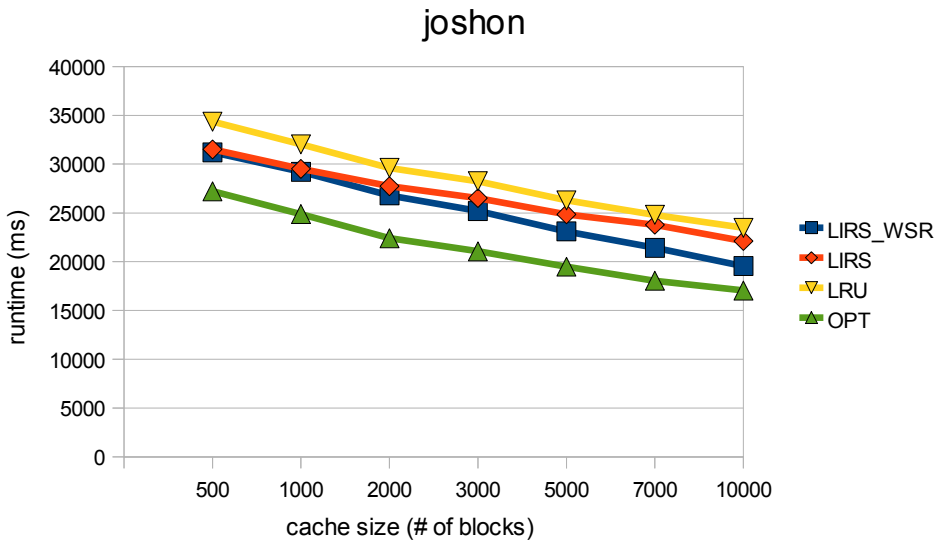


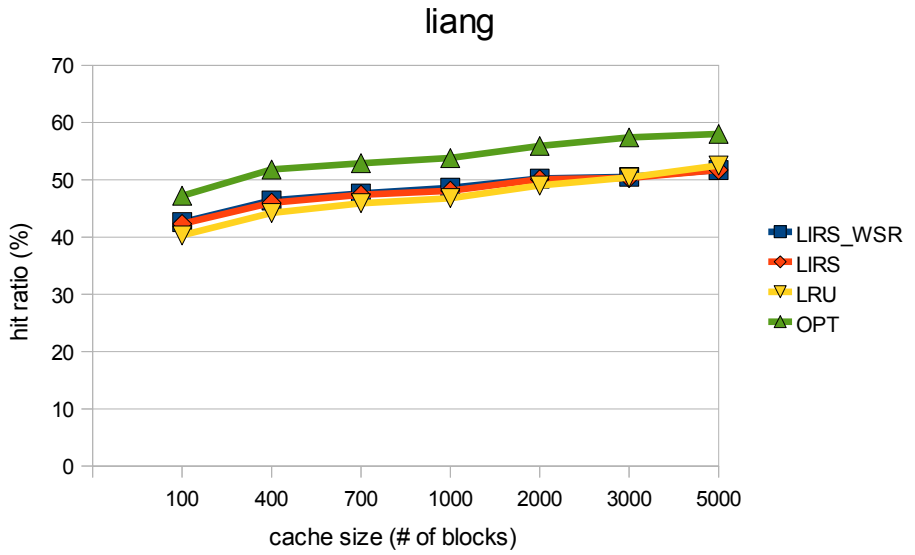


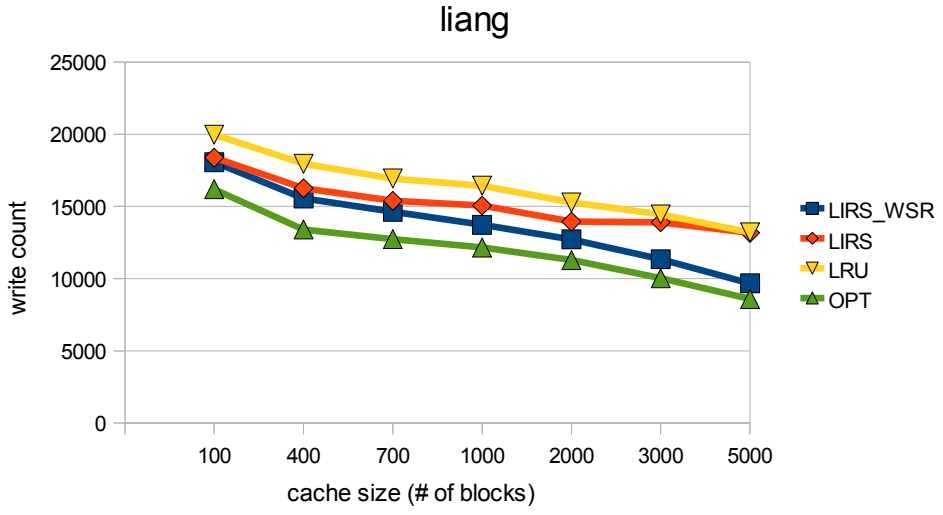

chang
runtime (ms)
350000
300000
250000
200000
150000
100000
50000
0
500 1000 2000 3000 5000 7000 10000 20000 30000 40000
cache size (# of blocks)
LIRS-WSR
LIRS
LRU

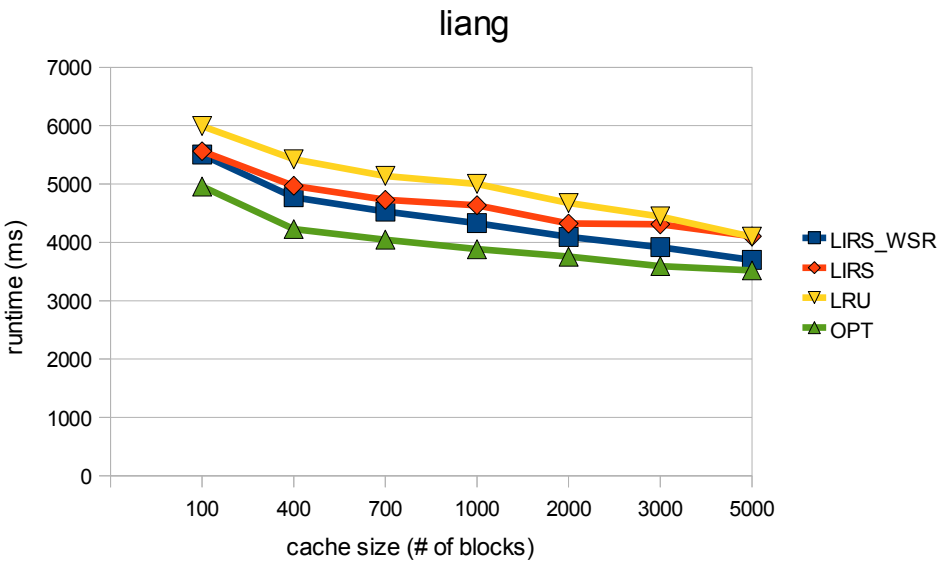

joshon
hit ratio (%)
80
70
60
50
40
30
20
10
0
500 1000 2000 3000 5000 7000 10000
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT


joshon
write count
140000
120000
100000
80000
60000
40000
20000
0
500 1000 2000 3000 5000 7000 10000
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT

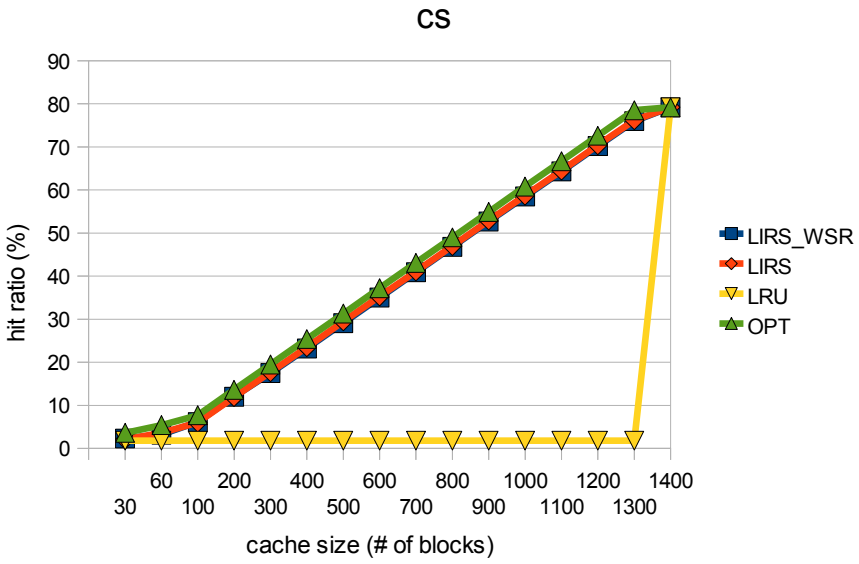

joshon
runtime (ms)
40000
35000
30000
25000
20000
15000
10000
5000
0
500
1000
2000
3000
5000
7000
10000
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT

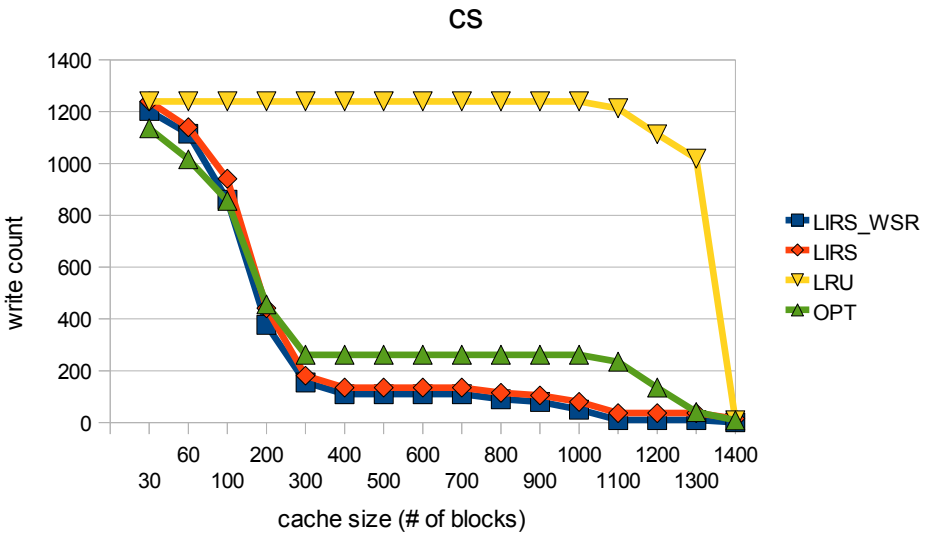

liang
hit ratio (%)
70
60
50
40
30
20
10
0
100
400
700
1000
2000
3000
5000
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT

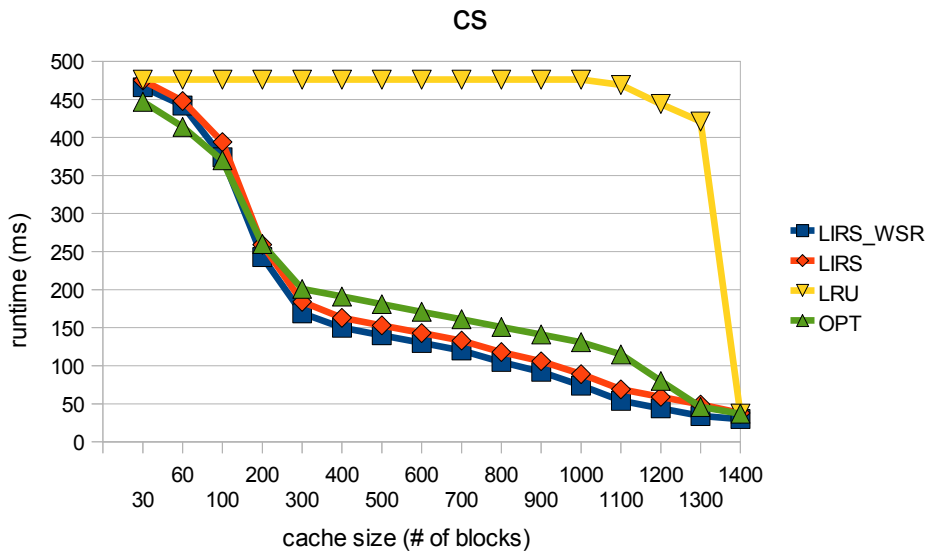

liang
write count
25000
20000
15000
10000
5000
0
100
400
700
1000
2000
3000
5000
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT

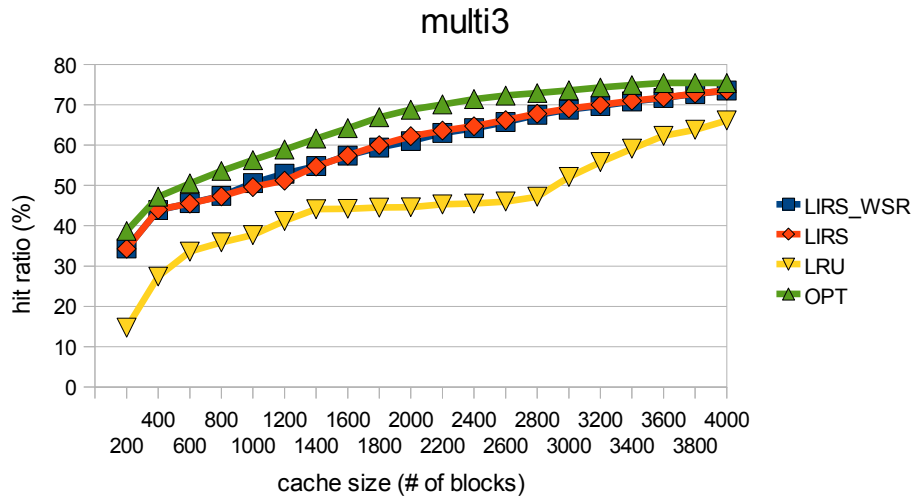


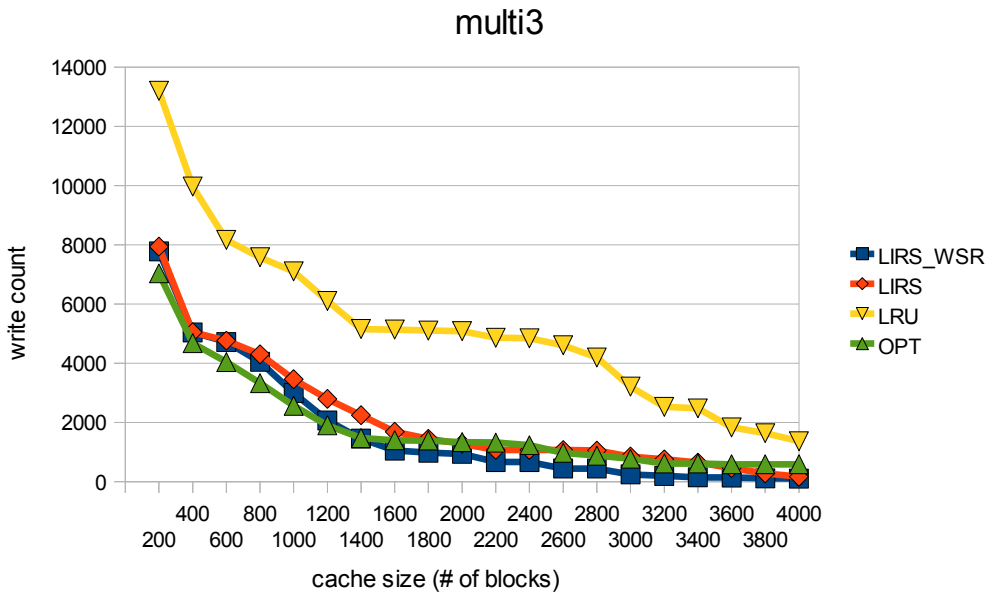
Επίσης, χρησιμοποιήσαμε τα αρχεία traces cs και multi3 του Song Jiang. Πιο συγκεκριμένα, το cs είναι ένα C source program examination tool trace, το οποίο πήρε ως είσοδο ένα αρχείο C μεγέθους 9 MB, και το multi3 είναι δημιουργήθηκε εκτελώντας τέσσερις εφαρμογές : cpp, gnuplot, glimpse και postgres. Το cpp είναι ένας GNU C compiler preprocessor, το gnuplot είναι ένα εργαλείο δημιουργίας γραφημάτων, το glimpse είναι ένα εργαλείο text information retrieval utility και το postgres είναι ένα μια ένωση επερωτήσεων ανάμεσα σε τέσσερις σχέσεις σε ένα σύστημα σχεσιακής βάσης δεδομένων από το πανεπιστήμιο της California στο Berkley.

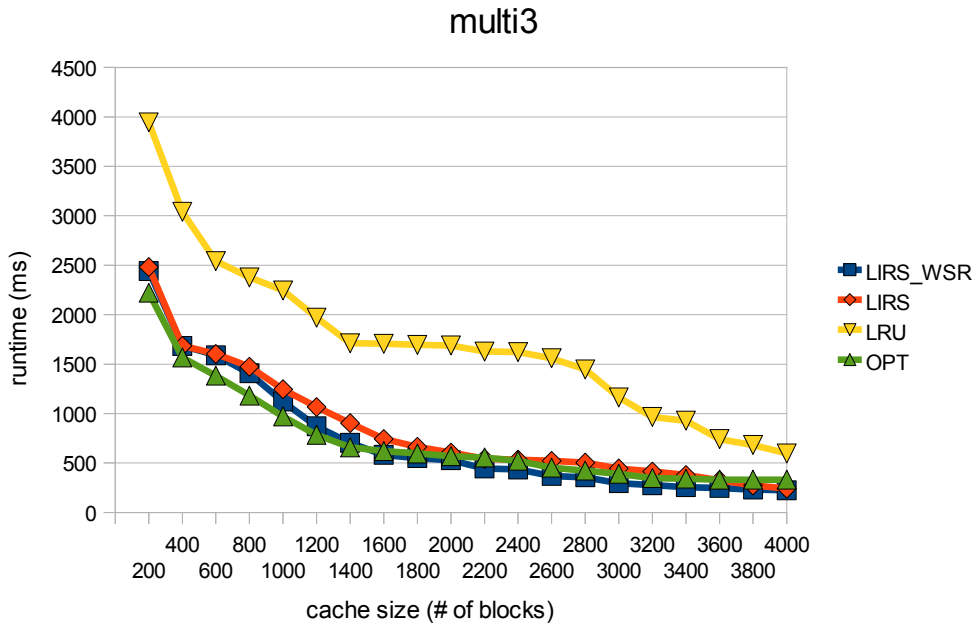



cs
write count
1400
1200
1000
800
600
400
200
0
60
200
400
600
800
1000
1200
1400
30
100
300
500
700
900
1100
1300
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT


cs
runtime (ms)
500
450
400
350
300
250
200
150
100
50
0
60
200
400
600
800
1000
1200
1400
30
100
300
500
700
900
1100
1300
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT


multi3
hit ratio (%)
80
70
60
50
40
30
20
10
0
400
800
1200
1600
2000
2400
2800
3200
3600
4000
200
600
1000
1400
1800
2200
2600
3000
3400
3800
cache size (# of blocks)
LIRS_WSR
LIRS
LRU
OPT





Στο σημείο αυτό, παρουσιάζουμε τα πολύ ενδιαφέροντα αποτελέσματα των πειραμάτων που πραγματοποίησαν οι σχεδιαστές του LIRS-WSR, στα οποία πραγματοποιούν σύγκριση ανάμεσα στον LRU, τον CFLRU, τον LIRS, τον ARC και τον LIRS-WSR.

Τα χαρακτηριστικά των αρχείων traces που χρησιμοποίησαν εμφανίζονται στους παρακάτω πίνακες.

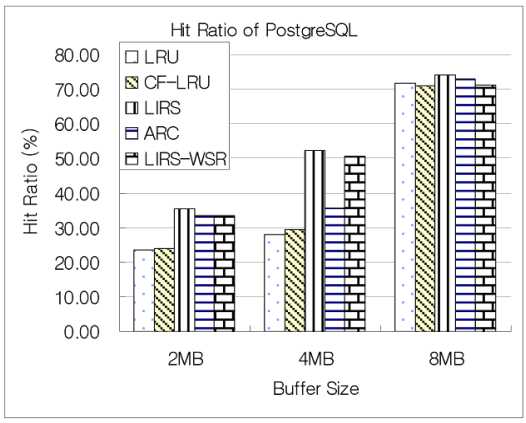
Η έκφραση τοπικότητας p% / g% σημαίνει ότι το g% των συνολικών προσβάσεων αφορούν το p% του συνολικού αριθμού σελίδων.

File System	YAFFS
Applications	Wisconsin Benchmark
Physical Page Size	512 Bytes
Logical Page Size	4 Kbytes
Total # of I/O Requests	51893
Total # of Page Write	5751 (11.08 %)
Read Locality	30% / 70%
Write Locality	15% / 85%

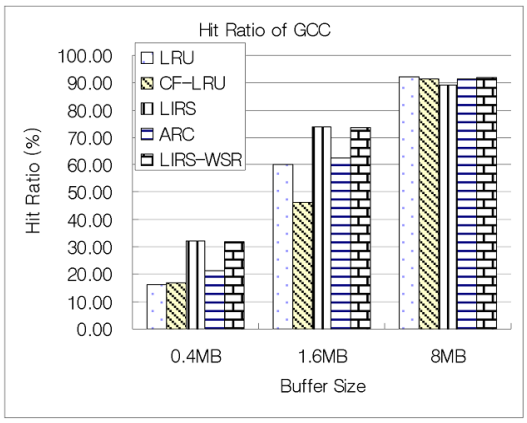
Πίνακας 6.4 : τα χαρακτηριστικά του PostgreSQL

Application	gcc builds on Linux	Viewperf benchmark on Linux OS	Cscope Tool On Linux
Logical Page Size	4 Kbytes	4 Kbytes	4 Kbytes
Total # of I/O Requests	158667	303123	202590
Total # of Writes Req.	19088 (12.03 %)	7333 (2.42%)	11057 (5.46%)
Read Locality	12% / 88%	33% / 67%	41%/59%
Write Locality	`32% / 68%	38% / 62%	25%/75%

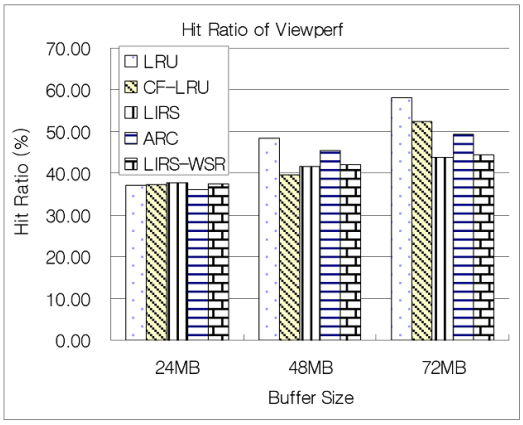
Πίνακας 6.5 : τα χαρακτηριστικά του gcc, Viewperf και Cscope



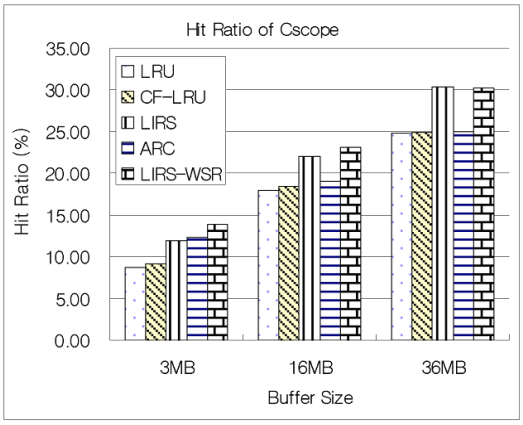
(a) PostgreSQL



(b) gcc

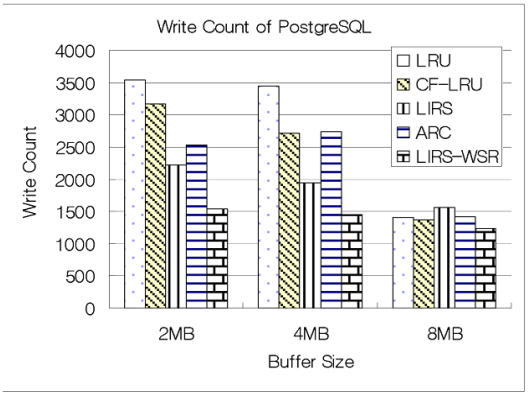


(c) Viewperf

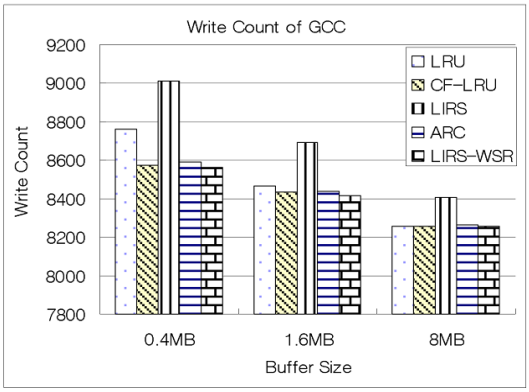


(d) Cscope

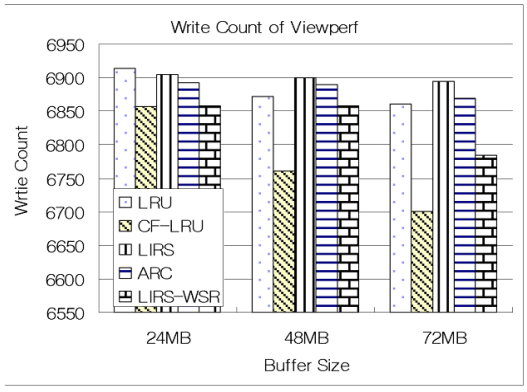
Όπως προαναφέρθηκε, αν η τοπικότητα στις εγγραφές είναι χαμηλή, όπως για παράδειγμα στην περίπτωση του Viewperf, η πολιτική WSR μπορεί να μην είναι αποτελεσματική και ακόμα και να μειώσει τη συνολική απόδοση, καθώς το όφελος από τη μείωση των εγγραφών μπορεί να είναι μικρότερο από το επιπλέον κόστος που παράγεται εξαιτίας της αύξησης των αναγνώσεων που προκαλείται από την υποβάθμιση του ποσοστού επιτυχίας στην cache.



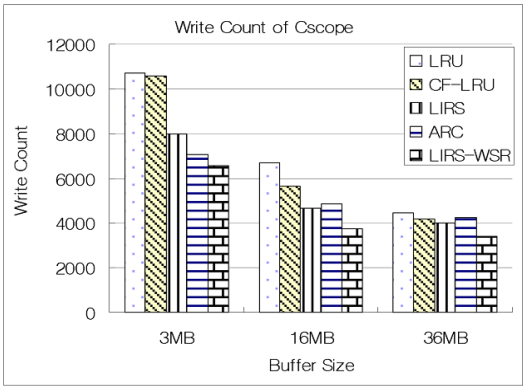
(a) PostgreSQL



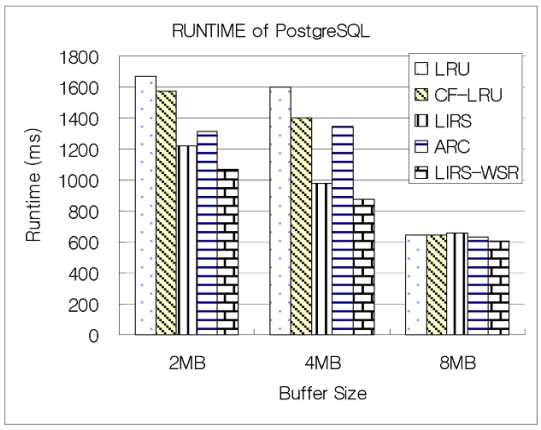
(b) gcc



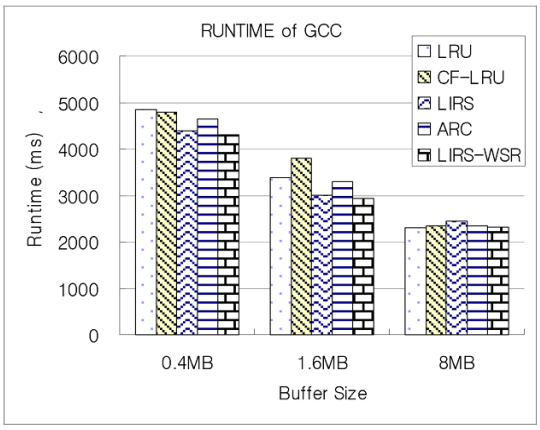
(c) Viewperf



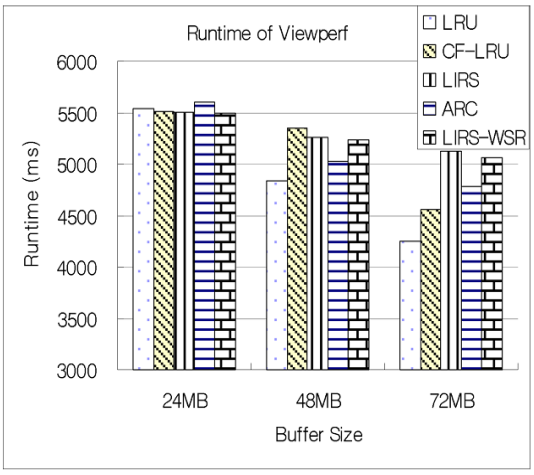
(d) Cscope



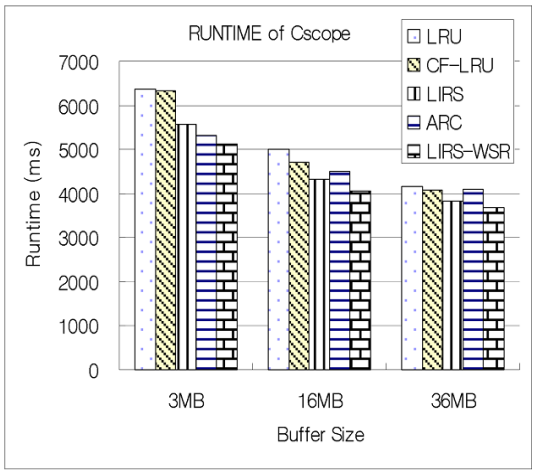
(a) PostgreSQL



(b) gcc



(c) Viewperf



(d) Cscope

7. Συμπεράσματα και μελλοντική εργασία

Στη μνήμη Flash, μια λειτουργία εγγραφής είναι πολύ πιο αργή και χρονοβόρα από μια λειτουργία ανάγνωσης, ενώ μια λειτουργία διαγραφής είναι πιο χρονοβόρα από μια λειτουργία εγγραφής. Η μείωση του πλήθους των εγγραφών στη μνήμη Flash, μπορεί να επιδεινώσει τη συνολική απόδοση του συστήματος εξαιτίας της ελάττωσης του ποσοστού επιτυχίας την buffer cache. Με άλλα λόγια, το όφελος που προκύπτει από τη μείωση των εγγραφών στη μνήμη Flash πρέπει να είναι μεγαλύτερο από το επιπλέον κόστος των επιπρόσθετων λειτουργιών ανάγνωσης, που προκαλείται από τη μείωση του ποσοστού επιτυχίας στην buffer cache, ούτως ώστε να βελτιωθεί η συνολική απόδοση του συστήματος.

Οι παραδοσιακοί αλγόριθμοι αντικατάστασης buffer δεν λαμβάνουν υπόψη τους τα χαρακτηριστικά της μνήμης Flash, ιδιαίτερα τις ασύγχρονες καθυστερήσεις ανάγνωσης και εγγραφής. Οι αλγόριθμοι αυτοί έχουν σχεδιαστεί για συστήματα όπου η δευτερεύουσα αποθήκευση είναι ένας σκληρός δίσκος και ως κύριο μέλημα έχουν τη μεγιστοποίηση του ποσοστού επιτυχίας στην buffer cache. Δυστυχώς όμως, αυτό μόνο δεν αρκεί για να φανούν αποτελεσματικοί και ανταγωνιστικοί για τη μνήμη Flash.

Ο αλγόριθμος αντικατάστασης buffer για μνήμη Flash, LIRS-WSR δημιουργήθηκε από την ένωση του παραδοσιακού αλγόριθμου αντικατάστασης LIRS και της πολιτικής WSR (Write Sequence Reordering). Ο LIRS χρησιμοποιεί εκτός από το πόσο συχνά προσπελάστηκε μια σελίδα (recency), και τη συχνότητα χρησιμοποίησης της (frequency), με αποτέλεσμα να υπερέχει τις περισσότερες φορές του αλγορίθμου LRU. Η πολιτική WSR επαναδιατάσει τις εγγραφές των ζεστών και ταυτόχρονα βρόμικων σελίδων, με σκοπό να μειώσει τον αριθμό των εγγραφών στη μνήμη Flash. Παράλληλα, για να αποφευχθεί η παραμονή των κρύων σελίδων στον buffer, ο αλγόριθμος χρησιμοποιεί την τεχνική cold-page detection, για να αναγνωρίζει αν μια σελίδα είναι ζεστή ή κρύα.

Υλοποιήσαμε τον LIRS-WSR στη γλώσσα προγραμματισμού C και προσομοιώσαμε τη λειτουργία του χρησιμοποιώντας αρχεία traces από τυπικά patterns πρόσβασης. Η προσομοίωση επαληθεύει τη μεγάλη βελτίωση της απόδοσης και την υπεροχή του LIRS-WSR έναντι άλλων αλγορίθμων, κυρίως του LRU. Επιπλέον, η μείωση του αριθμού των εγγραφών από τον LIRS-WSR, εκτός από καλύτερες επιδόσεις, έχει ως αποτέλεσμα την αύξηση της διάρκειας ζωής της μνήμης Flash.

Όσον αφορά το μέλλον, θα ήταν πολύ χρήσιμη η αξιολόγηση του LIRS-WSR κάτω από πραγματικές συνθήκες, δηλαδή η υλοποίηση του σε πραγματικό υπολογιστικό σύστημα. Επίσης, μια πιθανή ενίσχυση του LIRS-WSR με την τεχνική Block-Padding ίσως βελτίωνε ακόμα περισσότερο την απόδοση, καθώς με την τεχνική αυτή μεγιστοποιείται η πιθανότητα να πραγματοποιηθεί μια λειτουργία switch merge, έναντι μιας λειτουργίας full merge. Επιπλέον, η διατήρηση βρόμικων σελίδων στον buffer μπορεί να δημιουργήσει πρόβλημα ακεραιότητας για το σύστημα αρχείων, αν διακοπεί ξαφνικά η τροφοδοσία ηλεκτρικού ρεύματος, αφού η μνήμη RAM είναι πτητική. Βέβαια, το πρόβλημα αυτό είναι γενικευμένο και δεν αφορά μόνο τον LIRS-WSR, αλλά όλους τους αλγόριθμους αντικατάστασης buffer. Τέλος, ένα πολύ

σημαντικό θέμα για μελλοντική μελέτη είναι οι αδυναμίες των τρεχόντων σχημάτων log-block FTL και ιδιαίτερα το πρόβλημα του λυγισμού των log blocks και του υψηλού βαθμού συσχέτισης τους.

8. Βιβλιογραφία

[1]. Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. ***CFLRU: a replacement algorithm for flash memory.***

[2]. Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. ***FAB: Flash-Aware Buffer Management Policy for Portable Media Players***

[3]. Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, Jaehyuk Cha. ***LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory***

[4]. Hyojun Kim and Seongjun Ahn. Software Laboratory of Samsung Electronics, Korea. ***BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage***

[5]. Dongyoung Seo and Dongkun Shin. ***Recently-Evicted-First Buffer Replacement Policy for Flash Storage Devices***

[6]. Song Jiang and Xiaodong Zhang. ***Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance***

[7]. Song Jiang and Xiaodong Zhang. ***LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance***

[8]. Hoyoung Jung, Kyunghoon Yoon, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. ***LIRS-WSR: Integration of LIRS and Writes Sequence Reordering for Flash Memory***

[9]. Taeho Kgil and Trevor Mudge. ***FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers***

[10]. Intel Corporation. ***Understanding the Flash Translation Layer (FTL) Specification***

[11]. Li-Pin Chang and Tei-Wei Kuo. ***An Adaptive Stripping Architecture for Flash Memory Storage Systems of Embedded Systems***