Ατομική Διπλωματική Εργασία

FSort: Αλγόριθμος εξωτερικής ταξινόμησης για Ασύρματες Συσκευές Αισθητήρων με μνήμη τύπου Flash

Ορέστης Σπανός

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιούνιος 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

FSort: Αλγόριθμος εξωτερικής ταξινόμησης για Ασύρματες Συσκευές Αισθητήρων με μνήμη τύπου Flash

Ορέστης Σπανός

Επιβλέπων Καθηγητής
Γιώργος Σαμάρας

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου.

Ιούνιος 2009

Ευχαριστίες

Πρώτα από όλα θα ήθελα να ευχαριστήσω Τον Θεό που με αξίωσε να ολοκληρώσω αυτή τη Ατομική Διπλωματική Εργασία και κατ' επέκταση να τελειώσω με επιτυχία τις σπουδές στο Τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου. Επίσης θα ήθελα να ευχαριστήσω τους ανθρώπους που με την εμπειρία και τις γνώσεις τους με βοήθησαν να φέρω εις πέρας αυτή την εργασία. Ονομαστικά θα ήθελα να ευχαριστήσω τον Δρ. Γιώργο Σαμάρα που ήταν ο επιβλέπων καθηγητής αυτής της δουλειάς. Επίσης θα ήθελα να εκφράσω τις ιδιαίτερες ευχαριστίες μου στο Δρ. Δημήτρη Ζεϊναλιπούρ που ήταν ο εμπνευστής του θέματος και καθοδηγητής σε όλες τις φάσεις ανάπτυξης της εργασίας αυτής. Δεν θα μπορούσα να αφήσω πίσω τον μεταπτυχιακό φοιτητή του τμήματος κ. Παναγιώτη Ανδρέου που ήταν ο άνθρωπος που ερχόμασταν σε συχνή επαφή για όποιο πρόβλημα και δυσκολία αντιμετώπισα. Τέλος θα ήθελα να ευχαριστήσω τους συνεργάτες μου στο Εργαστήριο Κινητού και Ασύρματου Υπολογισμού για την άριστη συνεργασία μας που είχαμε σε όλη τη διάρκεια του έτους.

Θα ήταν άδικο να μην αναφέρω τις ευχαριστίες μου σε κάποια πολύ αγαπημένα μου πρόσωπα που παρόλο τους στερήθηκα και με στερήθηκαν, με ανέχτηκαν αυτή τη περίοδο που τους είχα ανάγκη.

Ευχαριστώ πολύ,

Ορέστης Σπανός

Περίληψη

Η εξωτερική ταξινόμηση είναι ένας όρος για μια κατηγορία αλγόριθμων ταξινόμησης που μπορούν να χειριστούν μεγάλες ποσότητες δεδομένων. Η εξωτερική ταξινόμηση απαιτείται όταν το μέγεθος των δεδομένων είναι μεγαλύτερο από την Κύρια Μνήμη (ΚΜ) συνήθως την RAM και πρέπει να χρησιμοποιηθεί ένα πιο αργό μέσο αποθήκευσης π.χ. Σκληρός Δίσκος.

Σε αυτή την Ατομική Διπλωματική Εργασία θα προτείνουμε, θα υλοποιήσουμε και θα αξιολογήσουμε ένα αλγόριθμο εξωτερικής ταξινόμησης που προτείνουμε, τον FSort ο οποίος θα σχεδιαστεί προσεχτικά ώστε να προνοεί τους περιορισμούς της μνήμης τύπου Flash και να εκμεταλλεύεται τα ιδιαίτερα χαρακτηριστικά τους ώστε να προσφέρει τη μέγιστη απόδοση σε μνήμες τύπου Flash.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

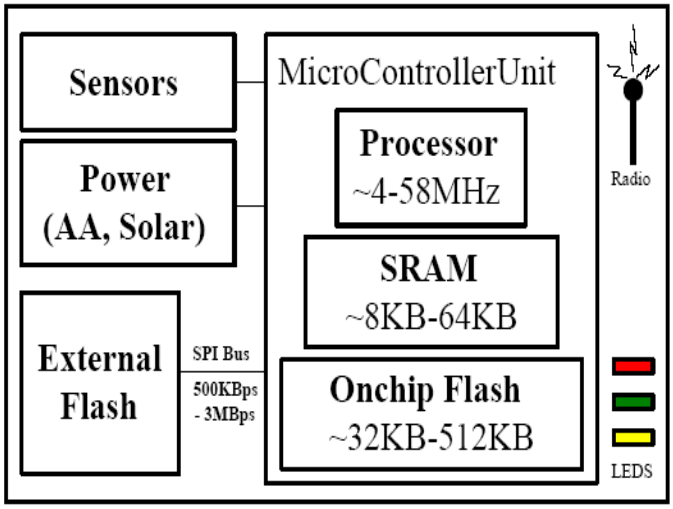
1.1 Εισαγωγή

Οι βελτιώσεις στη σχεδίαση υλικού μαζί με την ευρεία διαθεσιμότητα των οικονομικά βιώσιμων ενσωματωμένων Συσκευών Αισθητήρων (SDs), καθιστούν πλέον εφικτό να αλληλεπιδρούμε και να καταλάβουμε το φυσικό κόσμο με μεγάλη ακρίβεια [1,2,3]. Επιπλέον η ανάπτυξη των υψηλού επιπέδου πρωτοκόλλων επικοινωνίας όπως το IEEE802.15.4 και ZigBee [4] που χρησιμοποιούν μικρά και χαμηλής κατανάλωσης ενέργειας ψηφιακά ραδιοκύματα επέτρεψαν την δημιουργία των Ασύρματων Συσκευών Αισθητήρων (WSDs) που επικοινωνούν ασύρματα δημιουργώντας έτσι την Ασύρματα Δίκτυα Αισθητήρων(WSNs). Αυτό μαζί με την εξέλιξη των μέσων αποθήκευση τύπου Flash (Flash-based Storage) έχει επιτρέψει στους αισθητήρες την τοπική αποθήκευση και επεξεργασία των μαζικές ποσότητες μετρήσεων. Εκεί που στην αρχή οι συσκευές αισθητήρων διέθεταν 128-512 KB μνήμες τύπου flash –π.χ. ο Crossbow's MICAz (Εικόνα 1) διαθέτει 512KB - τώρα οι νεότεροι διαθέτουν πολύ πιο μεγάλη περιεκτικότητα π.χ. ο Crossbow's Imote2 διαθέτει 32MB [5].

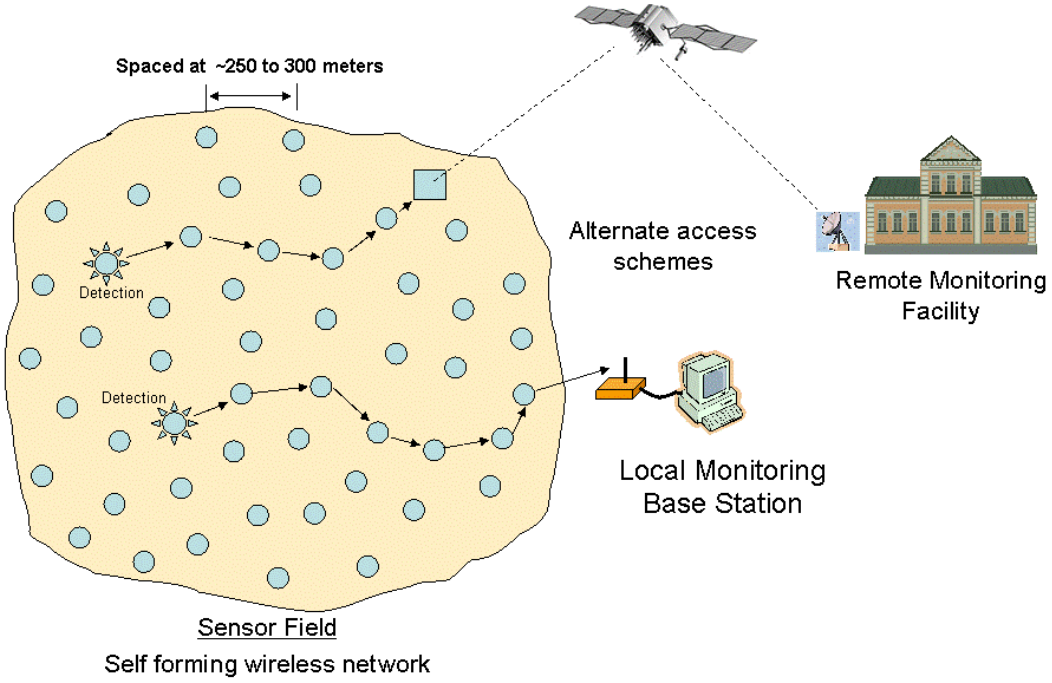
Εικόνα 1 Crossbow MICAz

Η αρχιτεκτονική ενός WSD φαίνεται στην Εικόνα 2. Όπως βλέπουμε υπάρχει η κεντρική μονάδα επεξεργασίας που αποτελείται από τον επεξεργαστή (Processor), τη Κύρια Μνήμη (SRAM) και την Onchip Flash. Επίσης βλέπουμε τους διάφορους αισθητήρες και την πηγή ενέργεια που μπορεί να είναι οι μπαταρίες ή ακόμα και η ηλιακή ενέργεια. Μετά μπορούμε να δούμε το Radio που χρησιμοποιείται για την ασύρματη επικοινωνία και τέλος την εξωτερική μνήμη Flash που μπορεί να ενωθεί με το MCU μέσω του SPI Bus.

Για τους σκοπούς αυτής της διπλωματικής μας ενδιαφέρουν δύο κομμάτια από την αρχιτεκτονική. Πρώτο η Onchip Flash συνήθως είναι τύπου NOR αλλά προσφέρει μικρότερες χωρητικότητες και η External Flash που συνήθως είναι τύπου NAND που προσφέρει μεγαλύτερες χωρητικότητες.

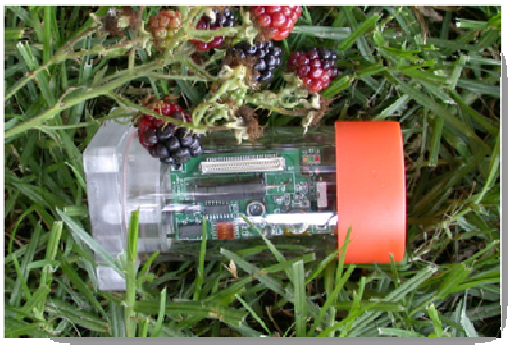


Εικόνα 2 Αρχιτεκτονική ενός WSD



Εικόνα 3 Ασύρματο Δίκτυο Αισθητήρων (WSN)

Τα WSNs (Εικόνα 3) μπορούν να χρησιμοποιηθούν σε πολυάριθμες εφαρμογές που κυμαίνονται μεταξύ πολλών από τον περιβαλλοντικό έλεγχο (όπως μετρήσεις ατμοσφαιρικής πίεσης και θερμοκρασίας βλ. Εικόνα 4) [6,3] μέχρι και σεισμικό ή δομικό έλεγχο βλ. Εικόνα 5 [7] καθώς επίσης και στον έλεγχο κατασκευής στην βιομηχανία [5,2].

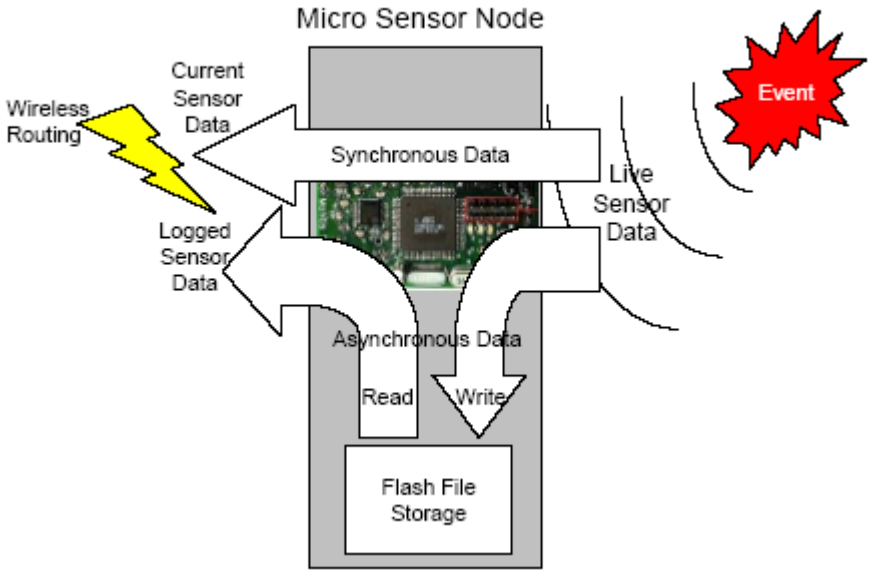


Εικόνα 4 Great Duck Island – Maine (Θερμοκρασία, Υγρασία)

Εικόνα 5 Έλεγχος δόνησης και μετατόπισης της δομής της γεφυρας Golden Gate – SF

Συνήθως αυτές οι εφαρμογές καταγράφουν συνεχώς τις μετρήσεις και τις διαβιβάζουν συνεχώς σε ένα σταθμό βάσης όπου εκεί τα δεδομένα επεξεργάζονται και παίρνονται οι ανάλογες αποφάσεις και αποτελέσματα (Synchronous Data). Εντούτοις σε μακροχρόνιες εφαρμογές συχνά προτιμείται να αποθηκεύονται τοπικά και να διαβιβάζονται μόνο όταν ζητηθούν από το χρήστη (Asynchronous Data). Οι δύο αυτές προσεγγίσεις φαίνονται στην Εικόνα 6. Αυτό είναι ιδιαίτερα καλό αφού η ασύρματη επικοινωνία είναι ενεργειακά πολύ πιο ακριβή από ρουτίνες αποθήκευσης [8] και

επεξεργασίας [2]. Συγκεκριμένα έχει αποδειχθεί ότι η κατανάλωση ενέργειας για διαβίβαση 1 bit δεδομένων χρησιμοποιώντας τον αισθητήρα MICA είναι περίπου ίση με την εκτέλεση 1000 εντολών KME [8].



Εικόνα 6 Synchronous Vs. Asynchronous Data

Αν και αποθηκεύοντας τα δεδομένα τοπικά είναι ιδιαίτερα ευεργετικό, ο τρόπος με τον οποίο έχουμε πρόσβαση στα δεδομένα δηλαδή οι δομές δεδομένων και οι αλγόριθμοι για τη αποθήκευση και ανάκτηση των δεδομένων είναι επίσης πολύ σημαντικό. Συχνά οι χρήστες τοποθετούν επερωτήσεις οι οποίες η επίδοση εκτέλεσης τους επηρεάζεται από τον τρόπο που είναι οργανωμένα τοπικά. Ιδιαίτερα οι εφαρμογές που εκτελούν επερωτήσεις διαστήματος (range queries), π.χ. "Βρες τους χρόνους όπου η θερμοκρασία ήταν μεταξύ 60^{o}C και 70^{o}C", μπορούν πραγματικά να ωφεληθούν από την ύπαρξη ταξινομημένων δεδομένων. Επιπλέον υπάρχουν περιπτώσεις όπου οι τεχνικές ενδοδικτυακών συναθροίσεων (in-network aggrecation/fusion) απαιτούν από τους διάφορους αισθητήρες γίνονται JOIN κάτω από μια συγκεκριμένη ιδιότητα. Τέτοιες JOIN διαδικασίες εκτελούνται γρηγορότερα όταν υπάρχουν τα δεδομένα είναι ταξινομημένα.

Η ταξινόμηση σε WSDs μπορεί να εκτελεσθεί με δύο τρόπους:

1. Online: Τα δεδομένα ταξινομούνται συνεχώς καθώς παίρνονται οι νέες μετρήσεις από τους αισθητήρες.

2. Offline: Τα δεδομένα αποθηκεύονται στην flash και ταξινομούνται περιοδικά κατόπιν αιτήσεως.

Η ερώτηση που εγείρεται είναι ποια από τις δύο τεχνικές θα μας προσφέρει την καλύτερη επίδοση στα WSDs. Για να απαντήσουμε αυτή την ερώτηση θα πρέπει πρώτα να ερευνήσουμε και να καταλάβουμε τα ιδιαίτερα χαρακτηριστικά μιας μνήμης flash που χρησιμοποιείται στα WSDs.

1.2 Χαρακτηριστικά μνήμης τύπου Flash

Η Flash memory ανήκει στη κατηγορία των electrically erasable programmable read-only memory γνωστή ως EEPROM. Είναι μόνιμη μνήμη (διατηρεί τα δεδομένα και χωρίς ηλεκτρικό ρεύμα), για αυτό χρησιμοποιείται για αποθήκευση αρχείων και άλλων δεδομένων σε σταθμούς εργασίας, εξυπηρετητών καθώς και υπολογιστές χειρός, κινητών τηλεφώνων, αισθητήρες κλπ.

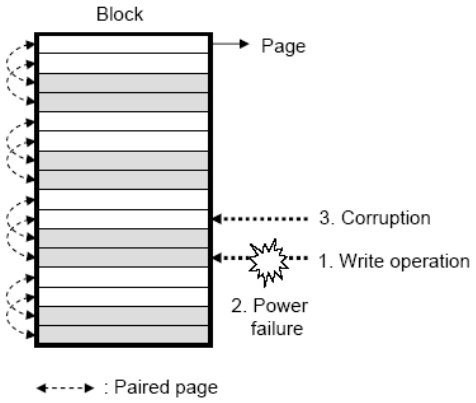
Η συμπεριφορά του flash memory στις κύριες λειτουργίες read/write/erase είναι ριζικά διαφορετικές από αυτές που ξέρουμε στα άλλα μέσα μόνιμης αποθήκευσης όπως το κυριότερο, οι μαγνητικοί σκληροί δίσκοι. Μια από τις σημαντικότερες διαφορές είναι ότι το κάθε memory cell σε μια flash memory έχει περιορισμένο αριθμό εγγραφής , μεταξύ 10.000 και 1.000.000, τις οποίες αν ξεπεράσει τότε φθείρονται και γίνονται αναξιόπιστα (Wear).
Υπάρχουν δύο κατηγορίες flash memory, οι Nor και οι Nand, που διαφέρουν αρκετά μεταξύ τους. Και στις δύο κατηγορίες, οι λειτουργίες write μπορούν να καθαρίσουν (clear) bits δηλαδή να αλλάξει η τιμή τους από 1 σε 0. Ο μόνος τρόπος να κάνεις set τα bits (αλλάξει η τιμή τους από 0 σε 1) είναι να διαγράψεις (erase) μια ολόκληρη περιοχή που ονομάζεται erase unit. Ένα erase unit έχει σταθερό μέγεθος συνήθως από μερικά μέχρι εκατοντάδες KB.

Οι Nor flash, που είναι πιο παλιός τύπος, είναι μια τυχαία προσπελάσιμη συσκευή που είναι προσβάσιμη απευθείας από τον επεξεργαστή. Κάθε bit μπορεί ανεξάρτητα να

καθαριστεί (1 σε 0) μια φορά σε κάθε κύκλο διαγραφής του erase unit στο οποίο ανήκει. Οι μνήμες αυτές υποφέρουν από μεγάλους χρόνους διαγραφής [3].

Οι Nand flash, ο νεώτερος τύπος, έχουν πολύ γρηγορότερους χρόνους διαγραφής, αλλά δεν είναι απευθείας προσβάσιμοι από τον επεξεργαστή (υπάρχει controller ο οποίος έχει πρόσβαση και δέχεται εντολές). Το διάβασμα γίνεται σε pages και όχι σε bit ή byte. Τα erase units που ανάφερα πιο πάνω χωρίζονται σε pages (συνήθως 512B). Ένα page μπορεί να τροποποιηθεί μόνο αν διαγραφεί το ολόκληρο το erase unit στο οποίο ανήκει. Έτσι όταν θέλουμε να τροποποιήσουμε ένα sector συνεχώς θα πρέπει να αντιγράψουμε όλα τα άλλα pages που ανήκουν σε αυτό το erase unit, να διαγράψουμε όλο το erase unit, και ακολούθως να γράψουμε ξανά όλο erase unit με ανανεωμένο το page που θέλουμε. Οι Nand flash χωρίζονται σε 2 κατηγορίες την Single-Level Cell (SLC) και την Multi-Level Cell (MLC).

Οι SLC αποθηκεύουν ένα bit σε κάθε cell σε αντίθεση με τις MLC που αποθηκεύουν περισσότερα του ενός bit σε κάθε cell χρησιμοποιώντας διαφορετική ένταση ρεύματος π.χ. οι 2-bit MLC χρησιμοποιούν 4 διαφορετικές εντάσεις ρεύματος. Λόγω του ότι οι MLC προσφέρουν μεγαλύτερη χωρητικότητα σε λιγότερο κόστος, χρησιμοποιούνται τώρα πια ευρέως και υπολογίζεται ότι θα αντικαταστήσει σε πολλές συσκευές το σκληρό δίσκο.



Εικόνα 7

Οι MLC από την άλλη έχουν και κάποια μοναδικά χαρακτηριστικά εκτός από αυτά που είδαμε μέχρι τώρα. (2-4KB) και όχι sector. Επίσης τα Το ελάχιστο μέγεθος εγγραφής είναι το page pages πρέπει να γραφτούν σε διαδοχική σειρά μέσα σε ένα block (erase

unit) δηλαδή μετά την εγγραφή του ιοστου page, το j(0<j<i) page δεν μπορεί να γραφτεί μέχρι να διαγραφεί και πάλι ολόκληρο το block. Επίσης αν διακοπεί το ρεύμα κατά τη διάρκεια ενός write σε ένα page τότε μπορεί να επηρεαστούν και άλλες σελίδες από το ίδιο block (paired page problem) (Εικόνα 7). Αυτό γίνεται γιατί τα bits που αποθηκεύονται σε ένα cell δεν είναι του ίδιου page αλλά διαφορετικών page. Οι σελίδες που μοιράζονται τα ίδια cell ονομάζονται paired pages [9].

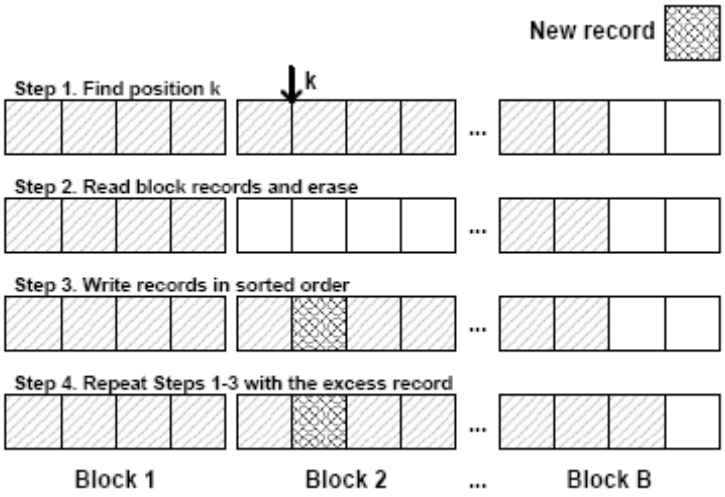
Λόγω αυτών των ιδιαιτεροτήτων, οι τεχνικές για την αποθήκευση που έχουν σχεδιαστεί για άλλους τύπους μνήμης, όπως οι μαγνητικοί δίσκοι, δεν είναι πάντα κατάλληλες για flash. Έτσι από τις αρχές του 1990 όπου εμφανίστηκαν ευρέως οι μνήμες flash έχουν αναπτυχθεί τεχνικές συγκεκριμένα για flash. Κάποιες από αυτές έχουν αναπτυχθεί συγκεκριμένα για flash, ενώ πολλές από αυτές είναι προσαρμογές των τεχνικών για άλλους τύπους μνήμης.

Πίνακας 1 Ενδεικτικοί παράμετροι επίδοσης για NAND-based flash memory χρησιμοποιώντας 3.3V voltage, 512B Page size και 16KB Block size

NAND-based flash memory installed on a WSD			
	Page Read	**Page Write**	**Block Erase**
	1.17mA	37mA	57mA
Time	6.25ms	6.25ms	2.26ms
Energy	24μJ	763μJ	425μJ
	Page Erase – Write	**Flash Idle**	**Flash Sleep**
	43mA	0.068mA	0.031mA
Time	6.75ms	N/A	N/A
Energy	957μJ	220μJ/sec	100μJ/sec

1.3 Online VS Offline Sorting

Τώρα που έχουμε δει τα ιδιαίτερα χαρακτηριστικά και περιορισμούς της μνήμης Flash θα απαντήσουμε ποια από τις δύο προσεγγίσεις είναι καλύτερη με τη εκτέλεση κάποιας προκαταρκτικής ανάλυσης.



Εικόνα 8 Παράδειγμα Online Sorting

Τώρα θα εξετάσουμε το παράδειγμα στην Εικόνα 8. Το παράδειγμα παρουσιάζει μια μνήμη flash που αποτελείται από *B* Blocks το καθέ ένα από αυτά περιλαμβάνει 4 *P=4* Pages. Σε μια χρονική στιγμή *r* η flash περιλαμβάνει *N* στοιχεία σε ταξινομημένη σειρά. Αν υποθέσουμε ότι τη χρονική στιγμή *r+1* ένα νέο στοιχείο έρχεται και πρέπει να αποθηκευτεί και αυτό και να παραμείνει η ταξινομημένη σειρά. Το πρώτο βήμα είναι να βρεθεί η θέση *k* στο οποίο πρέπει να εισαχθεί το νέο στοιχείο. Αυτό χρειάζεται *k* reads. Όταν βρεθεί το *k*, το block το οποίο περιλαμβάνει το *k* διαβάζεται (P reads) και γίνεται erased λόγω του Erase-Constraint ούτως ώστε να γίνει update. Αν το block περιλαμβάνει περισσότερο από ένα άδεια pages τότε το στοιχείο εισάγεται και ακολούθως ξαναγράφεται το Block. Αντίθετα αν το block δεν έχει άδεια pages τότε η διαδικασία συνεχίζεται στο επόμενο block με το επιπλέον page μέχρι να βρεθεί κάποιο block που διαθέτει κάποιο page άδειο. Αυτή η τεχνική είναι εξαιρετικά επικίνδυνη για την flash αφού αποτελείται από πάρα πολλές πράξεις writes/erases τα οποία μειώνουν την απόδοση και τη διάρκεια ζωής της μνήμης. Στη πραγματικότητα θα δείξω αυτό το επίπεδο ζημιάς χρησιμοποιώντας μια σειρά benchmarks που θα παρουσιαστούν στο κεφάλαιο Πειραματική Αξιολόγηση.

Από την άλλη, το offline sorting απαιτεί μόνο ότι τα δεδομένα είναι ταξινομημένα μόνο όταν ζητείται. Αυτό είναι γενικά καλύτερο σε περιπτώσεις όπου γράφοντας στη flash είναι σημαντικά ακριβότερο από την ανάγνωση όπως στην μνήμη flash. Επίσης γίνονται πιο λίγα erases που δημιουργούν το πρόβλημα της αναξιοπιστίας (wear). Ο

αλγόριθμος FSort που προτείνουμε είναι σχεδιασμένος ώστε να επεκτείνει τις βασικές αρχές του offline sorting για να αυξήσει περισσότερο της απόδοση του με την ελαχιστοποίηση των writes/erases. Αυτό παρέχει αποδοτική πρόσβαση στα στοιχεία που αποθηκεύονται στη flash καθώς επίσης και την μακροζωία της με το να κατανέμει ομοιόμορφα τα write/erases σε όλα τα pages ώστε η διαθέσιμη χωρητικότητα να μην μειώνεται σε ορισμένες περιοχές της flash.

Κεφάλαιο 2

Σχετική Εργασία

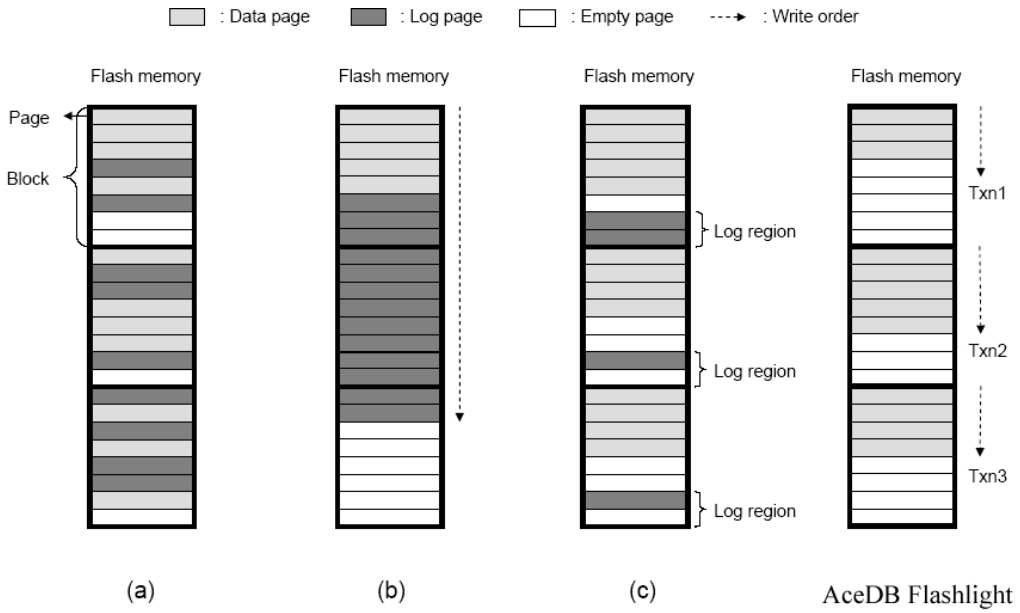


Fig. 4. Comparison of flash-based database approaches. (a) FTL approach. (b) Log-structured approach. (c) In-page logging approach.

Εικόνα 9

Τώρα θα εξετάσουμε 4 προσεγγίσεις για βάσεις δεδομένων σε flash memory, το Flash Translation Layer, το Log-Structured Approach, το In-Page Logging Approach (IPL) και τέλος το AceDB Flashlight. Θα εξετάσουμε πως δουλεύουν σε γενικές γραμμές καθώς και που υστερούν η κάθε μια.

2.1 Block-Mapping Techniques (Flash Translation Layer - FTL)

Μια προσέγγιση να χρησιμοποιηθεί η μνήμη flash είναι να την χειριζόμαστε σαν μια συσκευή με blocks σταθερού μεγέθους που μπορούν να διαβαστούν και να γραφτούν όπως ακριβώς τα sectors σε ένα σκληρό δίσκο. Αυτό επιτρέπει να χρησιμοποιήσουμε τη μνήμη flash με τα ήδη υπάρχοντα file systems. Έτσι ο κώδικας των file system καλεί μεθόδους στον driver της συσκευής ζητώντας λειτουργίες read/write. Το driver της flash memory είναι υπεύθυνο για το πώς θα γίνουν αυτές οι λειτουργίες.
Όμως, το πως θα γίνουν αυτά τα read/write δεν είναι τόσο απλό όσο σε ένα σκληρό δίσκο αφού εμπεριέχει τους περιορισμούς που είδαμε πιο πάνω. Πρέπει να προσέξουμε να μην υπάρχουν blocks στη flash που να διαγραφούν πολύ περισσότερες φορές από άλλα. Επίσης το άλλο είναι για να γράψουμε κάτι στη flash πρέπει εκεί να είναι ήδη διαγραμμένο και διαγραφή γίνεται μόνο σε erase unit και όχι σε επίπεδο bytes. Είναι φανερό όμως ότι αν λύσουμε το πρώτο (wear-leveling) θα λυθεί αυτόματα και το δεύτερο.

Η βασική ιδέα πίσω από όλες τις wear-leveling τεχνικές είναι το block number που ξέρει το file system (virtual block number) να κάνει map στη φυσική διεύθυνση στη flash (sector). Αυτό λέγετε virtual-block-to-sector map. Όταν ένα virtual block θα ξαναγραφτεί, τα νέα δεδομένα δεν θα γραφτούν στον ίδιο χώρο που ήταν τα παλιά αλλά σε ένα ήδη διαγραμμένο sector και θα ανανεωθεί το virtual-block-to-sector map. Συνήθως τα sectors είναι υποδιαίρεση του erase unit και έχουν μέγεθος μιας σελίδας. Αυτή η τεχνική προσπαθεί να λύσει τα δύο προβλήματα που ανάφερα στην προηγούμενη παράγραφο καθώς και να προφέρει ατομικότητα στα writes. Αν το ρεύμα αποκοπεί κατά την διάρκεια της εκτέλεσης του write τότε δεν χάνουμε δεδομένα και

μπορούμε να επανέλθουμε στην κατάσταση που ήμασταν αφού δεν διαγράψαμε τα παλιά δεδομένα [10].

2.2 Log-Structured Approach

Μια άλλη προσέγγιση είναι να έχουμε πρόσβαση απευθείας στη flash χωρίς τη χρήση κάποιου επιπλέον επιπέδου όπως το FTL. Η λογική σε αυτή τη τεχνική είναι να χειριζόμαστε ολόκληρη τη βάση δεδομένων σαν ένα μεγάλο log. Όταν γίνει ανανέωση κάποιου αντικειμένου τότε εισάγουμε μια εγγραφή log χωρίς να αλλάξουμε το αντικείμενο. Έτσι οι εγγραφές γίνονται συνεχόμενες πάντα στο τέλος της ΒΔ. Στα θετικά αυτής της προσέγγισης είναι ότι προσφέρει καλή απόδοση στις εγγραφές (writing). Όμως για να βρούμε την τιμή ενός αντικειμένου θα πρέπει να διαβάσουμε πολλά log pages. Επίσης ούτε αυτή η προσέγγιση λαμβάνει υπόψη της καθόλου το paired-page problem [11].

2.3 In-Page Logging Approach (IPL)

Σε αυτή την προσέγγιση, τα log records που σχετίζονται με αντικείμενα που βρίσκονται σε κάποιο block αποθηκεύονται στο ίδιο αυτό block. Έτσι μπορούμε να αναδημιουργήσουμε αποδοτικά ένα database page με το να διαβάσουμε μόνο τα pages στο ίδιο block. Σε ένα block καθορίζουμε από πριν πόσα και ποια pages θα χρησιμοποιούμε για αντικείμενα και πόσα και ποια pages θα χρησιμοποιηθούν για logging. Αυτό βελτιώνει τη απόδοση στο writing αφού εισάγουμε και πάλι μόνο log εγγραφές αντί να αλλάζουμε τα database pages. Το ότι τα log files είναι διασκορπισμένα στη μνήμη δεν μας επηρεάζει αφού μπορούμε όπως είπαμε και πιο πάνω δεν έχουμε επιπλέον κόστος στο read λόγω μετακίνησης κινούμενων εξαρτημάτων στη flash και όλα τα pages διαβάζονται με το ίδιο χρονικό κόστος (25μs) όπου και αν βρίσκονται.

Όμως το ότι τα log pages είναι προκαθορισμένα σε κάθε block, αν γεμίσουν τότε θα πρέπει να βρεθεί νέο block άδειο στο οποίο θα πάνε κάποια database pages και όλα τα log records που αφορούν αυτά τα αντικείμενα. Επίσης όπως και πιο πριν, στις MLC flash τα άδεια pages θα τελειώσουν πιο γρήγορα αφού το ελάχιστο μέγεθος εγγραφής είναι το page.Τέλος ούτε αυτή η προσέγγιση λαμβάνει υπόψη το paired-page problem [12].

2.4 AceDB Flashlight

Το AceDB Flashlight για να εξαλείψει τα πιο πάνω προβλήματα δεν θα χρησιμοποιήσει καθόλου log records που είναι συχνές μικρές εγγραφές αφού το να γράψουμε μικρά log δεν έχει διαφορά με το να γράψουμε ολόκληρο page στη flash. Έτσι αντί να γράφει log θα γράφει το ανανεωμένο page ολόκληρο αν χρειάζεται. Επίσης μετατρέπει τα τυχαία λογικά write από ένα transaction σε φυσικά συνεχόμενα writes σε κάποιο block. Τέλος λαμβάνει υπόψη του το page paired problem και έτσι προσφέρει αξιοπιστία στα δεδομένα σε περίπτωση αποκοπής του ηλεκτρικού ρεύματος.

Χρησιμοποιεί logical-to-physical map και όταν γίνει μια αλλαγή σε κάποιο αντικείμενο σε κάποιο page αυτό το page γράφετε σε ένα άδειο χώρο και αλλάζει κατάλληλα το logical-to-physical page map στην RAM. Αυτός ο μηχανισμός θυμίζει shadow mapping. Έτσι δεν έχουμε logging αλλά μπορούμε να κάνουμε κανονικά recovery αφού τα παλιά δεδομένα δεν σβήνονται.

Επίσης, σε αντίθεση με όλες τις τεχνικές που ανάφερα πιο πάνω, όταν χρειάζεται να γίνει allocate ένα άδειο block λαμβάνεται υπόψη λαμβάνεται υπόψη το transaction το οποίο θέλει να κάνει το write και όλα τα υπόλοιπα write θα γίνουν από το ίδιο transaction στο ίδιο block, όπως δείχνει η Εικόνα 9 για τα 3 transaction txn1, txn2, txn3. Έτσι λύνεται το pair-page problem γιατί αν έχουμε απώλεια ρεύματος σε κάποιο write αυτό επηρεάζει μόνο pages για το ίδιο transaction. Έτσι το transaction αυτό θα γίνει abort και άρα δεν μας νοιάζει που τα δεδομένα εκεί δεν είναι reliable. Επίσης λόγω του ότι πάντα γίνονται συνεχόμενα writes έχουμε πολύ καλή απόδοση σε αυτό τον τομέα όπως στο log-structured approach.

2.5 Micro-Hash: An efficient index structure for flash-based sensor devices

Αυτό που προσπαθεί να λυθεί με το microhash είναι να μειώσει τις διαγραφές που κοστίζουν στις μνήμες flash. Επίσης να προσφέρει γρήγορες αναζητήσεις τόσο ισότητας π.χ. αν αποθηκεύουμε μετρήσεις θερμοκρασίας σε κάποιους χρόνους, να βρίσκουμε σε ποιους χρόνους η θερμοκρασία ήταν 30C (value-based equality queries), όσο και να πάρουμε τις μετρήσεις σε ένα χρονικό εύρος π.χ. να πάρουμε τις μετρήσεις που πήραμε από το 10:00 μέχρι τις 10:30 (time-based range and equality queries). Επίσης λήφθηκαν υπόψη οι φυσικοί περιορισμοί που έχουν οι flash όπως αυτές εξήγησα στο πρόλογο μου, δηλαδή διαμοιράζει τις εγγραφές σε όλη τη μνήμη ώστε να αποφύγει το wearing σε κάποιες σελίδες που τυχόν γράφονται πιο συχνά (Wear-Leveling), Ελαχιστοποιεί τις διαγραφές σελίδων αφού για να γίνουν πρέπει να διαγραφούν όλες οι σελίδες στο erase block που αυτή ανήκει (Block-Erase) και ελαχιστοποιεί το μέγεθος της κυρίως μνήμης RAM που χρειάζονται για δομές δεδομένων για το indexing.

Μπορεί να χρησιμοποιηθεί σε sensor networks και γενικά όπου θέλουμε να αποθηκεύουμε δεδομένα, συνήθως σταθερού μεγέθους ανά τακτά χρονικά διαστήματα, και να μπορούμε να κάνουμε queries στα δεδομένα αυτά [8].

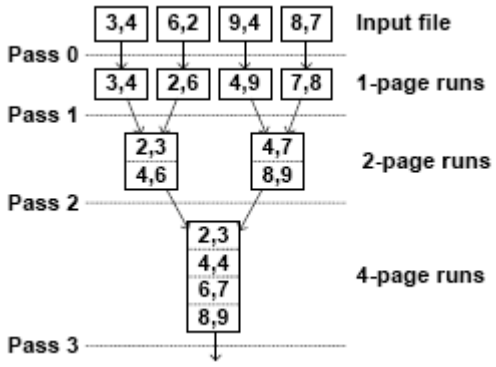
2.6 External Sorting

Η ταξινόμηση είναι ένα από τα θεμελιώδη προβλήματα της πληροφορικής και έχει μελετηθεί εκτενώς από την ερευνητική κοινότητα. Η εξωτερική ταξινόμηση προκύπτει όταν το μέγεθος των δεδομένων που θα ταξινομηθούν είναι μεγαλύτερος από την διαθέσιμη κύρια μνήμη του συστήματος. Οι βασικές αρχές της εξωτερικής ταξινόμησης έχουν αναλυθεί λεπτομερώς στο [13] χρησιμοποιώντας tapes σαν αποθηκευτικό μέσο. Από τότε, έχει γίνει πολλή ερευνητική δουλειά για την ενσωμάτωση και βελτίωση των

αλγορίθμων εξωτερικής ταξινόμησης (I/O Operations, απαιτήσεις μνήμης κλπ) στους σκληρούς δίσκους.

Οι παραδοσιακοί αλγόριθμοι εξωτερικής ταξινόμησης αποτελούνται συνήθως από δύο φάσεις:

1. Εσωτερική Ταξινόμηση: Σε αυτή τη φάση το αρχείο εισόδου μεγέθους *S* pages εισέρχεται στη Κύρια Μνήμη κατά κομμάτια τέτοια που να χωρούν στη ΚΜ, ταξινομούνται και γράφονται πίσω στη flash (Pass 0). Οι προκύπτουσες ταξινομημένες σελίδες ονομάζονται *runs*. Μόλις τελειώσει η φάση αυτή αρχίζει η δεύτερη φάση της Εξωτερικής Συγχώνευσης.
2. Εξωτερική Συγχώνευση: Ας υποθέσουμε ότι έχουμε διαθέσιμη ΚΜ μεγέθους *M* Pages, στη φάση αυτή χρησιμοποιεί ένα page από αυτά για σκοπούς output buffer και τις υπόλοιπες *M-1* για input buffers. Σε κάθε πέρασμα *M-1* runs συγχωνεύονται και παράγονται μεγαλύτερα runs.



Εικόνα 10 Παράδειγμα Εξωτερικής Ταξινόμησης με M=3 (2 input και 1 output page buffers)

Μια πολύ πρόσφατη εργασία ερευνά το που ερευνά τη εξωτερική ταξινόμηση σε κινητές συσκευές, ονομάζεται FAST, έχει δημοσιευθεί πρόσφατα στο [14]. Ο FAST μειώνει τον αριθμό των writes χρησιμοποιώντας σωρό μεγέθους *M-1* pages στη ΚΜ και ένα page για είσοδο/έξοδο. Το αρχείο εισόδου ανιχνεύεται συνεχώς προκειμένου να διατηρηθούν τα μικρότερα κλειδιά στο σωρό. Χρησιμοποιώντας αυτόν τον τρόπο, ο FAST απαλείφει την ανάγκη για την φάση της συγχώνευσης αφού παράγεται ένα και μόνο run σε ένα pass. Αυτή η προσέγγιση αυξάνει την γενική απόδοση και μειώνει το πρόβλημα της αναξιοπιστίας αλλά χρειάζεται πολύ περισσότερες ανιχνεύσεις του

αρχείου εισόδου. Επιπλέον οι συντάκτες υποθέτουν ότι υπάρχει αρκετή flash για να χωρέσει το αρχείο εξόδου.

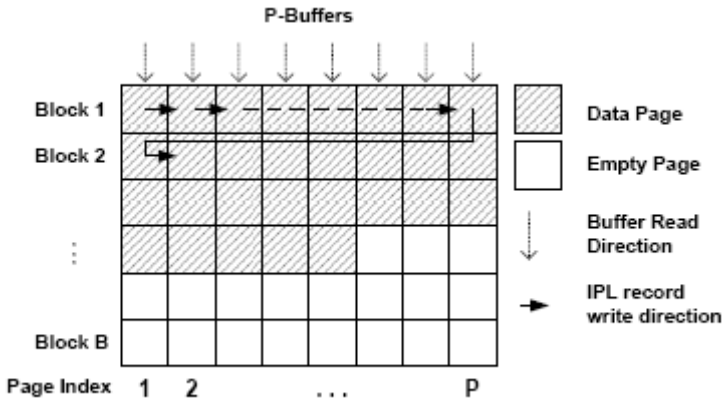
Κεφάλαιο 3

Μοντέλο Συστήματος

Πίνακας 2 Ορισμοί Συμβόλων

Σύμβολο	Ορισμός
N	Χωρητικότητα Flash
M	Χωρητικότητα KM (pages)
B	Πλήθος Blocks
b_i	Block με δείκτη i: $1 \leq i \leq B$
P	Αριθμός των Pages σε ένα Block
p_i^j	Page με δείκτη i στο block j

Σε αυτό το κεφαλαίο θα τυποποιήσουμε τις βασικές ορολογίες μας και υποθέσεις. Τα κύρια σύμβολα και οι αντίστοιχοι ορισμοί τους συνοψίζονται στο Πίνακας 2. Για συντομία θα αναφερόμαστε στη μνήμη NAND-based Flash Memory σαν Flash. Στη Εικόνα 11 βλέπουμε μια απλοϊκή αναπαράσταση της Flash. Η Flash περιέχει *B* Blocks ($b_1, b_2, .. b_B$) και κάθε ένα περιλαμβάνει *P* pages. Καθορίζουμε p_i^j σαν το page με δείκτη *i:* $1 \leq i \leq P$ που ανήκει στο block *j* : $1 \leq j \leq B$. Επίσης υποθέτουμε ότι ο WSD διαθέτει KM μεγέθους *M* pages.



Εικόνα 11 Μοντέλο Συστήματος

Σε αυτή τη Ατομική Διπλωματική Εργασία υιοθετούμαι μια τροποποιημένη έκδοση της IPL προσέγγισης όπου οι νέες εγγραφές που προκύπτουν από τις συνεχόμενες επερωτήσεις διατηρούνται σε ένα buffer. Μόλις το buffer γεμίσει τότε γράφεται διαδοχικά στο τέλος της Flash π.χ. στο επόμενο διαθέσιμο κενό Page. Αργότερα θα παρουσιάσουμε γιατί ένα τέτοιο σενάριο είναι ευεργετικό. Η Εικόνα 11 παρουσιάζει τη κατεύθυνση των IPL writes.

Όπως περιγράφεται νωρίτερα στο Κεφάλαιο 1.3, ο αλγόριθμος FSort αποτελείται από δύο φάσεις, την εσωτερική ταξινόμηση και την εξωτερική συγχώνευση. Κατά τη διάρκεια της φάσης εσωτερικής ταξινόμησης δημιουργούνται *P* buffers και γίνεται η πρόσβαση τους κάθετα όπως δείχνει η Εικόνα 11 με τα κάθετα διακεκομμένα βέλη. Τα ολοκληρωμένα runs πάντα γράφονται στο τέλος της μνήμης flash.

Κεφάλαιο 4

Ο Αλγόριθμος FSort

Σε αυτό το κεφάλαιο θα περιγράψουμε τον δικό μας αλγόριθμο με όνομα FSort. Ο FSort αποτελείται από δύο φάσεις:

1. Φάση Εσωτερικής Ταξινόμησης
2. Φάση Εξωτερικής Συγχώνευσης

4.1 Φάση Εσωτερικής Ταξινόμησης

Αυτή η φάση παράγει ταξινομημένες ακολουθίες σε αύξουσα σειρά (sorted runs) που χρησιμοποιούνται σαν είσοδος στη δεύτερη φάση. Τα δεδομένα διαβάζονται και μεταφέρονται στη ΚΜ με κατακόρυφη σειρά και τα sorted runs γράφονται στο τέλος της Flash. Ο Αλγόριθμος για αυτή τη φάση είναι:

Algorithm 1 : FlashSort - Phase A

Input: D unsorted data pages, Available Flash Memory M, Buf:M-1 input buffers, min:1 output buffer
Output: D **sorted** data pages

```
1: procedure GENERATE_INITIAL_RUNS(S)
2:     Allocate(Buf(P)); //Allocate P page buffers for input
3:     CountEmpty=0; //Allocate P page buffers for input
4:     //Initiate the P-buffers
5:     for i = 1 to P do
6:         set Buf_j = p_i^1;
7:     end for
8:     while true do
9:         //find the page with the smallest key (min) and its
               index in Buf
10:        set (min, BIndex) = SelectionTree(Buf);
11:        Output(min); //write sorted output to flash
12:        //find the next page that the Buf_BInx must retrieve
13:        set PIndex = Buf_BIndex->header(pos)++;
14:        if (!IsEmpty(p_BIndex^PIndex) then //check if empty page
15:            set Buf_BIndex = p_BIndex^PIndex;
16:        else
17:            set CountEmpty++;
18:        end if
19:        if (CountEmpty == P) then
20:            Break;
21:        end if
22:    end while
23: end procedure
```

Η βασική ιδέα πίσω από αυτό το βήμα είναι να μειωθεί ο αριθμός των passes που θα χρειαστεί η δεύτερη φάση του αλγόριθμου ώστε να μειωθεί κατ' επέκταση τόσο η ενέργεια που θα καταναλωθεί για την διαδικασία της ταξινόμησης όσο και ο χρόνος που θα χρειαστεί να γίνει η ταξινόμηση. Η μείωση της ενέργειας θα γίνει αφού θα μειωθούν τα reads και writes. Αυτό είναι πολύ σημαντικό αφού θα γίνει η χρήση του σε WSDs που όπως έχω αναφέρει πιο πάνω έχει περιορισμένα αποθέματα ενέργειας που επηρεάζει το χρόνο που ένα WSD είναι ζωντανό.

Για να μειωθούν τα passes θα πρέπει να μειωθεί ο αριθμός των sorted runs που θα δημιουργηθούν. Η βασική διαφορά αυτής της φάσης σε σχέση με την πρώτη φάση του κλασσικού external merge sort είναι ότι παράγει όσο το δυνατό μεγαλύτερα sorted runs μεταβλητά σε μέγεθος. Αυτό δημιουργεί λιγότερα σε αριθμό sorted runs από τον βασικό αλγόριθμο. Με τον τρόπο που υλοποιείται η φάση αυτή τότε ο μέγιστος αριθμός sorted runs που θα δημιουργηθούν είναι $\left\lceil \frac{Total\ Data\ Pages}{Buffers\ Pages\ in\ RAM} \right\rceil$ που είναι ο ίδιο αριθμός με τα runs που παράγει ο external merge sort.

Ας υποθέσουμε ότι έχουμε *D* αταξινόμητα pages και διαθέσιμη Κύρια Μνήμη μεγέθους *M* pages. Τα M-1 buffers θα χρησιμοποιηθούν για input και ένα για output. Ο αλγόριθμος ξεκινά με την αρχικοποίηση των buffers με τα πρώτα M-1 pages από τη flash και αρχικοποίηση με 0 του μετρητή CountEmpty που θα χρησιμοποιηθεί για να τερματίσει τη διαδικασία δημιουργίας ενός sorted run αν δεν μπορεί να μεγαλώσει άλλο είτε λόγω ότι όλα τα pages χρησιμοποιήθηκαν είτε λόγω ότι δεν υπάρχει κάποιο record στη ΚΜ που να μπορεί να χρησιμοποιηθεί ώστε να μην χαλάσει η αύξουσα σειρά. Ακολούθως ο αλγόριθμος συνεχώς επιλέγει συνέχεια το πιο μικρό κλειδί (ή μεγαλύτερο ανάλογα αν θέλουμε φθίνουσα ταξινόμηση) συμπεριλαμβανομένου του buffer index, BIndex, δηλαδή ο δείκτης του buffer που έχει το μικρότερο κλειδί. Ακολούθως γράφεται στο output buffer. Η εύρεση του μικρότερου κλειδιού μπορεί να γίνει αποδοτικά αν χρησιμοποιήσουμε ένα selection tree. Ακολούθως πρέπει να φέρουμε στο Buf_{BIndex} την επόμενη κατάλληλη σελίδα δηλαδή M-1 σελίδες μετά την τελευταία σελίδα που διαβάσαμε για το συγκεκριμένο buffer. Το τελευταίο page που διαβάσαμε αποθηκεύεται στο header του Buf_{BIndex} στο πεδίο lastPageRead. Αν το επόμενο key είναι μικρότερο από αυτό που μόλις στείλαμε στο output ή η επόμενη σελίδα δεν ανήκει στα αρχικά δεδομένα τότε αυξάνουμε το CountEmpty κατά ένα. Μόλις το CountEmpty γίνει ίσο με M-1 τότε το sorted run ολοκληρώθηκε. Ακολούθως θέτουμε και πάλι τη τιμή 0 στο CountEmpty και ξεκινά το νέο run.

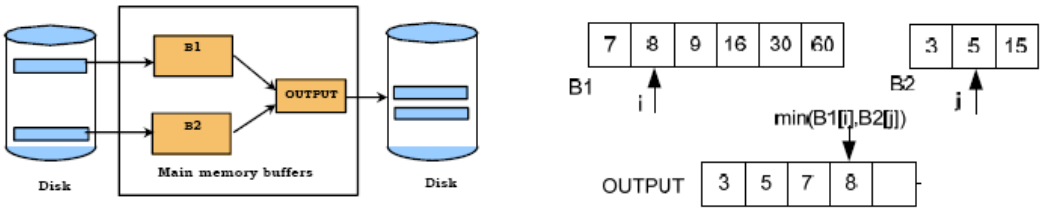
Η φάση αυτή ολοκληρώνεται όταν όλες οι σελίδες από το αρχείο εισόδου επεξεργαστούν. Μόλις τελειώσει αυτή η φάση ακολουθεί η δεύτερη φάση της εξωτερικής συγχώνευσης με είσοδο τα sorted runs από τη πρώτη φάση.

4.2 Φάση Εξωτερικής Συγχώνευσης

Κατά τη διάρκεια αυτής της φάσης, τα ταξινομημένα runs που έχουν παραχθεί στη φάση 1, συγχωνεύονται μέχρι να μείνει ένα και μόνο run που είναι και το τελικό αποτέλεσμα.

Στην Εικόνα 12 [15] ένα παράδειγμα εξωτερικής συγχώνευσης. Ο αλγόριθμος που γίνεται όπως περιγράφεται στο [15] είναι:

1. Load the available sorted runs R1 and R2 into main memory buffers B1 and B2 a page-at-a-time (i.e., initially first page from each run)
    - Obviously R1>=B1 and R2>=B2 (a Run might be larger than a Buffer)
    - The rest pages will be loaded to main memory during subsequent steps.
2. Initialize indices i, j to the head of each list (i.e., i=j=0)
3. Compare B1[i] with B2[j] and move smallest item to OUTPUT buffer.
    - If B1[i] was smallest item then i++ else j++ (see right figure)
    - If OUTPUT gets full, it is appended to the end of a file on DISK and cleared in RAM.
4. Repeat the above until either index i or j reaches the end of its buffer.
    - If i reached the end of B1 then load next available page from run R1, else load next available page from run R2 into B2. If neither option applies append remaining records to OUTPUT and finish.



Εικόνα 12 Φάση Εξωτερική Συγχώνευσης

Κεφάλαιο 5

Πειραματική Αξιολόγηση

Σε αυτό το κεφάλαιο θα παρουσιάσω αναλυτικά τις πειραματικές συγκρίσεις του αλγόριθμου FSort.

5.1 Flash Emulator

Έχω υλοποιήσει ένα Flash Emulator ο οποίος προσομοιώνει μια ρεαλιστική μνήμη flash. Είναι γραμμένος στη γλώσσα C. Ο flash emulator μπορεί να μας δώσει μετρήσεις όπως αριθμός reads, writes, erase ανά page καθώς και erases ανά blocks. Με αυτές τις μετρήσεις και σε συνδυασμό ένα πίνακα όπως π.χ. ο Πίνακας 1, μπορούμε να πάρουμε τη κατανάλωση ενέργειας καθώς και το χρόνο που αναλώθηκε σε I/O της μνήμης flash.

Οι βασικές συναρτήσεις που προσφέρει το flash emulator είναι:

```
bool flash_create(Flash_Memory* newFlashMem,
   char* filename,
   uint16_t B,
   uint16_t PPB,
   uint16_t BPP);
```

Η συνάρτηση αυτή δημιουργεί μια νέα εικονική flash με B Blocks, PPB pages ανά block και BPP Bytes ανά page. Η flash αποθηκεύεται στο δίσκο στο αρχείο με τον όνομα filename και το newFlashMem είναι δείκτης στη νέα εικονική flash που δημιουργήθηκε.
Επιστρέφει TRUE αν η δημιουργία έγινε με επιτυχία.

```
bool flash_open(Flash_Memory* fm,char* filename);
```

Ανοίγει μια εικονική flash που βρίσκεται στο αρχείο filename και το fm είναι δείκτης στην εικονική flash.
Επιστρέφει TRUE αν το αρχείο filename υπάρχει και περιέχει μια εικονική flash.

```
bool flash_import(Flash_Memory fm,
     uint32_t offset,
     char *sourceFilename);
```

Αποθηκεύει τα περιεχόμενα του αρχείου sourceFilename στη εικονική μνήμη fm. Το πρώτο byte που θα γραφτεί είναι στο byte με δείκτη offset.
Επιστρέφει TRUE αν μεταφορά του αρχείου έγινε με επιτυχία.

```
bool flash_export(Flash_Memory fm,
     uint32_t offset,
     uint32_t length,
     char* targetFilename);
```

Εξάγει τα length bytes ξεκινώντας από το byte offset που έχει η εικονική flash fm και τα αποθηκεύει στο αρχείο targetFilename.
Επιστρέφει TRUE αν η εξαγωγή των δεδομένων έγινε με επιτυχία.

```
bool flash_close(Flash_Memory fm);
```

Αφού αποθηκευτεί η κατάσταση της εικονικής flash fm στο δίσκο τότε κλείνουμε την εικονική flash και αποδεσμεύονται οι πόροι που χρησιμοποιεί.
Επιστρέφει TRUE αν το κλείσιμο έγινε με επιτυχία

```
bool flash_flush(Flash_Memory fm);
```

Αποθηκεύει την κατάσταση της εικονικής flash fm στο δίσκο.

Επιστρέφει TRUE αν η αποθήκευση έγινε με επιτυχία.

```
bool flash_readPage(void* buffer,
   uint32_t pageIndex,
   Flash_Memory fm);
```

Διαβάζεται το page με δείκτη pageIndex από την εικονική flash fm και αποθηκεύεται στο buffer.

Επιστρέφει TRUE αν η ανάγνωση του page έγινε με επιτυχία.

```
bool flash_writePage(void* buffer,
   uint32_t pageIndex,
   Flash_Memory fm);
```

Γράφει τα περιεχόμενα του buffer που έχουν μέγεθος όσο το μέγεθος ενός page, στο page με δείκτη pageIndex στη εικονική flash fm.

Επιστρέφει TRUE αν η εγγραφή του page έγινε με επιτυχία.

```
bool flash_eraseBlock(uint32_t blockIndex,
     Flash_Memory fm);
```

Διαγράφει τα περιεχόμενα του block με δείκτη blockIndex στη εικονική flash fm.

Επιστρέφει TRUE αν η διαγραφή του block έγινε με επιτυχία.

```
bool flash_erasePage(uint32_t pageIndex,
   Flash_Memory fm);
```

Διαγράφει τα περιεχόμενα του page με δείκτη PageIndex στη εικονική flash fm.

Επιστρέφει TRUE αν η διαγραφή του page έγινε με επιτυχία.

Η υλοποίηση της βιβλιοθήκης βρίσκεται στο Παράρτημα Α.

5.2 Datasets

Στα πειράματα που θα ακολουθήσουν χρησιμοποίησα ένα πραγματικό dataset που έχει συλλεχτεί από 58 αισθητήρες στις εγκαταστάσεις της Intel Research στο Berkeley μεταξύ 28 Φεβρουάριου και 5 Απριλίου 2004. Οι αισθητήρες που χρησιμοποιήθηκαν ήταν εξοπλισμένοι με καιρικά sensor boards και συνέλεξαν πληροφορίες τοπολογίας

καθώς επίσης και τιμές υγρασίας, θερμοκρασίας, φωτός και τάσης κάθε 31 δευτερόλεπτα. Το dataset περιλαμβάνει 2.3 εκατομμύρια εγγραφές.

5.3 Πειραματικά Αποτελέσματα Online vs. Offline Sorting on flash

Στην πρώτη πειραματική σειρά έχουμε κάνει ένα πείραμα ώστε να απαντήσουμε στο ερώτημα ποια είναι η καλύτερη τεχνική να ακολουθήσουμε για τον αλγόριθμο που θα προτείνουμε. Υλοποιήσαμε μια απλουστευμένη έκδοση του αλγόριθμου Insertion Sort και την παραδοσιακή έκδοση του External Merge Sort για να μετρήσουμε τη Online Sorting και Offline Sorting προσέγγιση αντίστοιχα. Στη περίπτωση του InsertionSort η ταξινόμηση γινόταν με το που ερχόταν μια εγγραφή ενώ στην περίπτωση του MergeSort έγινε αφού ήρθαν όλες οι εγγραφές και αποθηκεύτηκαν.

Δώσαμε στο simulator 1000 εγγραφές από το dataset που περιγράφετε στο 5.2 σε μια εικονική flash μεγέθους 512KB με μια εγγραφή ανά page για λόγους απλότητας και ταξινομήσαμε τα δεδομένα με βάση την τιμή του φωτός. Το simulator μας έδωσε το πόσα reads, writes και erases έγιναν και τις μετασχηματίσαμε σε κατανάλωση ενέργειας και χρόνου. Το simulator έτρεξε για 2 είδη flash: α)NAND-based Flash b)NOR-based Flash.

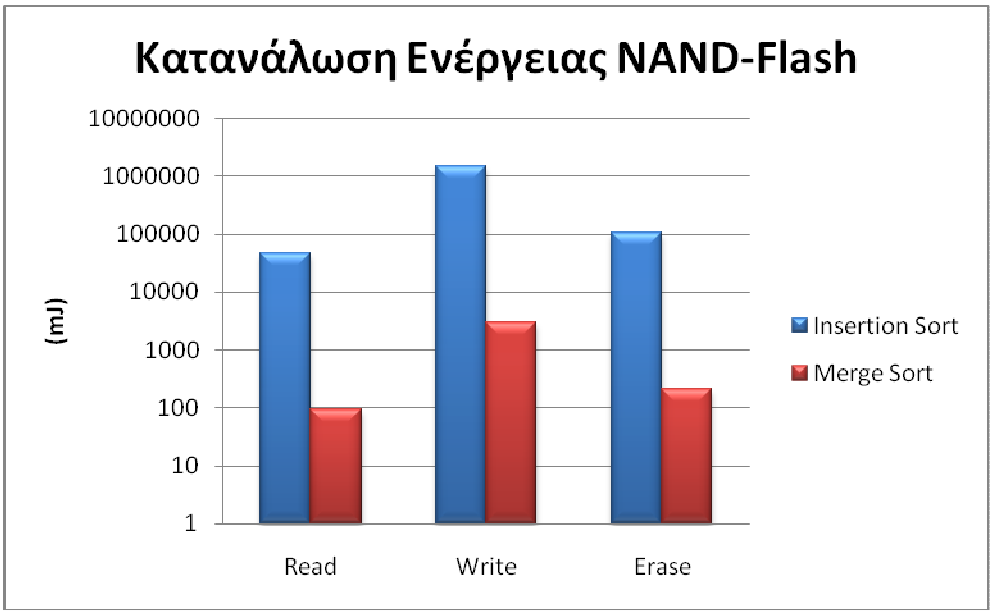
Τα αποτελέσματα τόσο για NOR-based flash όσο και για NAND-based flash φαίνονται στο Πίνακας 3. Όπως βλέπουμε η προσέγγιση του Offline sorting υπερνικά κατά πολύ την Offline και σε θέμα κατανάλωσης ενέργειας και σε θέμα χρόνου, και στους δύο τύπους flash. Συγκεκριμένα παρατηρούμε μια μείωση κατανάλωση ενέργειας περίπου τριών τάξεων μεγέθους στη NAND-Based και δύο τάξεων μεγέθους στη NOR-Based Flash. Επίσης ο InsertionSort κάνει πολλά erases και μάλιστα στα ίδια blocks που μπορεί να κάνουν τα block αυτά να γίνουν αναξιόπιστα. Ο merge sort κάνει πολύ πιο λίγα erases και μάλιστα σε όλα τα διαθέσιμα blocks που υπάρχουν (wear leveling). Ο InsertionSort έχει ένα πλεονέκτημα σε σχέση με τον Merge Sort. Δεν χρειάζεται επιπλέον χώρο στη flash για να γίνει η ταξινόμηση όπως τον merge sort που χρειάζεται

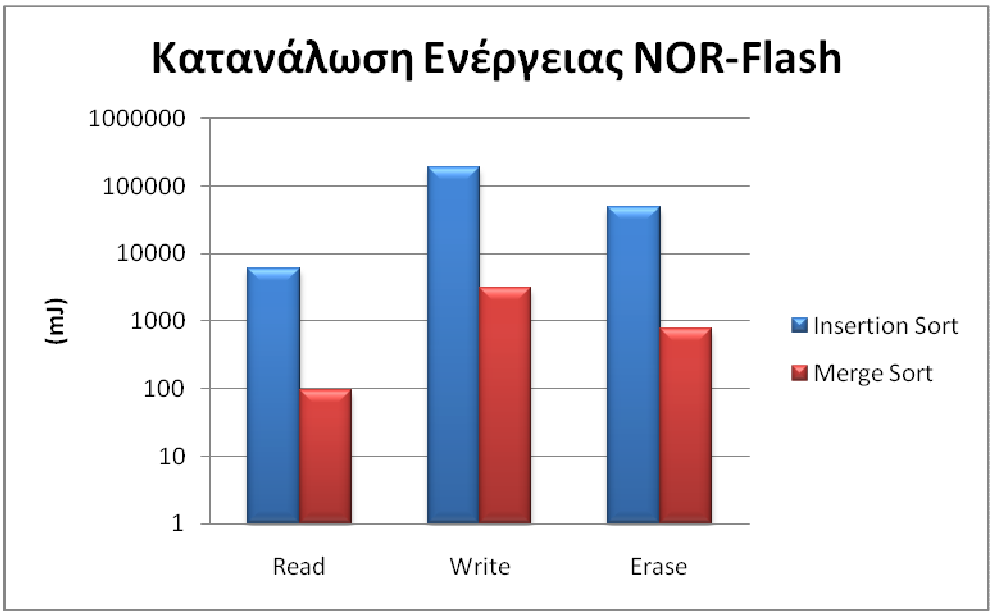
να υπάρχει διαθέσιμη χωρητικότητα όση η χωρητικότητα που έχουν τα δεδομένα εισόδου.

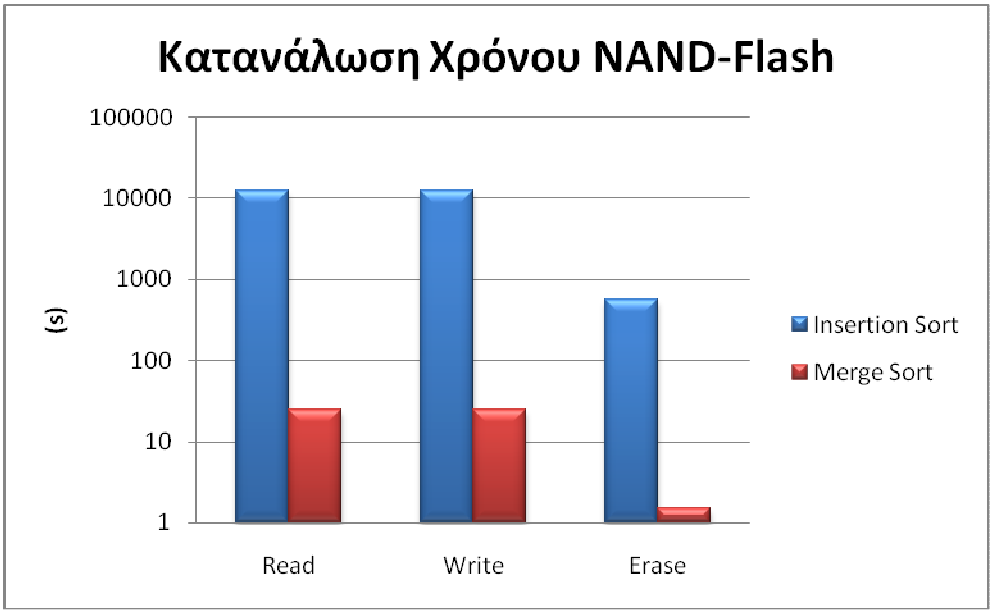
Πίνακας 3 Επίδοση Online vs. Offline Sorting

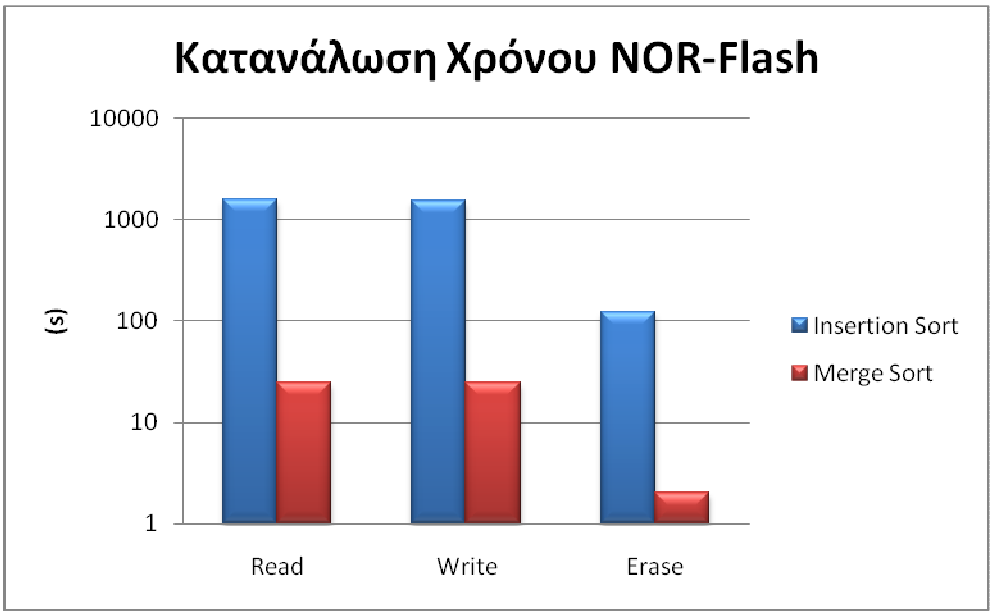
Ενέργεια (mJ)		**Read**	**Write**	**Erase**
NAND-flash	Insertion Sort	47815	1519376	105789
	Merge Sort	96	3052	213
NOR-flash	Insertion Sort	5998	189921	48279
	Merge Sort	96	3052	776
Χρόνος (s)				
NAND-flash	Insertion Sort	12452	12446	563
	Merge Sort	25	25	1,5
NOR-flash	Insertion Sort	1562	1546	122
	Merge Sort	25	25	2

Πιο κάτω φαίνονται οι γραφικές παραστάσεις για την κατανάλωση ενέργειας και χρόνου τόσο σε NAND Flash όσο και σε NOR Flash. Οι άξονες Y είναι λογαριθμικοί με βάση το 10 ώστε να φαίνεται εύκολα η διαφορά τάξης μεταξύ των δύο αλγορίθμων.




Κατανάλωση Ενέργειας NOR-Flash
1000000
100000
10000
1000
100
10
1
(mJ)
Read
Write
Erase
Insertion Sort
Merge Sort


Κατανάλωση Χρόνου NAND-Flash
100000
10000
1000
100
10
1
(s)
Read
Write
Erase
Insertion Sort
Merge Sort



5.4 Πειραματική Μέτρηση Επίδοσης του FSort

Σε αυτή τη πειραματική σειρά θα συγκρίνω την επίδοση του αλγόριθμου N-Way Megre Sort με τον FSort. Έχω υλοποιήσει τους δύο αυτούς αλγόριθμους σε γλώσσα C υπό τη μορφή simulation. Η σύγκριση της επίδοσης θα γίνει με βάση την κατανάλωση ενέργειας και χρόνου σε I/O και το wear leveling.

Δώσαμε στο simulator 32768 εγγραφές από το dataset που περιγράφεται στο 5.2. Οι εγγραφές εισάγονταν στο σύστημα μια-μια κάθε 30ms. Κάθε 30.72s δηλαδή κάθε 1024 εγγραφές γινόταν ταξινόμηση των δεδομένων. Τα πειράματα έγιναν για 3 χαρακτηριστικά: 1) Φως, 2)Θερμοκρασία και 3) Υγρασία, χρησιμοποιώντας 2 τύπους εικονικής flash: 1) NAND-Based και 2) NOR-Based. Για τις ανάγκες του simulation υποθέσαμε ότι στη RAM υπάρχουν διαθέσιμες 9 page buffers που χρησιμοποιήθηκαν και στους δύο αλγόριθμους ως εξής: 8 page buffers για είσοδο και 1 page buffer για έξοδο.

Πιο κάτω θα παρουσιάσουμε αναλυτικά τα αποτελέσματα που πήραμε τα simulation που έγιναν για το χαρακτηριστικό *"Φως"*. Ο Πίνακας 4 δείχνει τις μετρήσεις που πήραμε για την κατανάλωση ενέργειας και χρόνου σε flash τύπου NAND. Όπως βλέπουμε σε όλες τις μετρήσεις ο FSort έχει πιο χαμηλή κατανάλωση ενέργειας. Πιο

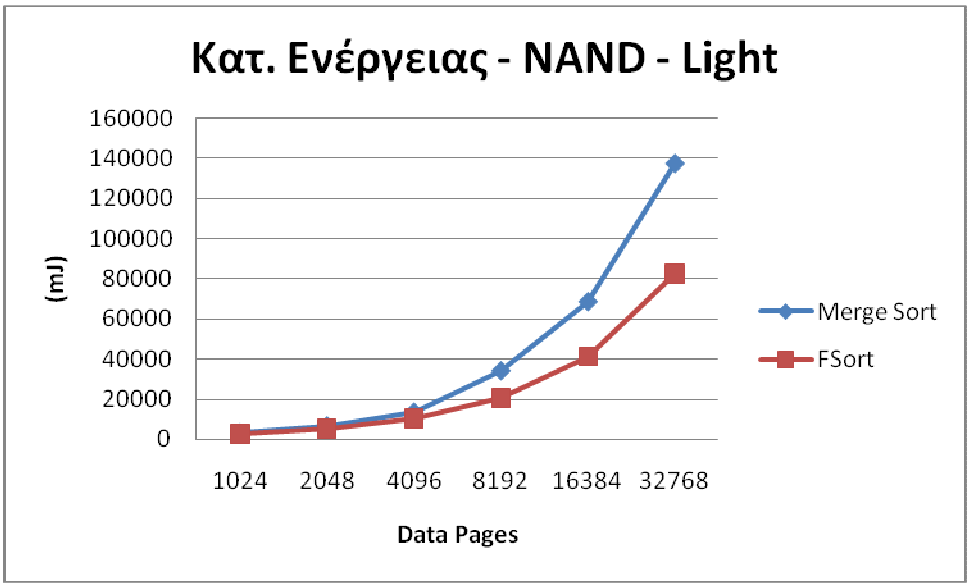
συγκεκριμένα βλέπουμε μέχρι τα 4096 records να έχουμε εξοικονόμηση περίπου 24% και μετά περίπου 41%. Ο Πίνακας 5 δείχνει τον συνολικό χρόνο που σπαταλήθηκε σε I/O από τους δύο αλγόριθμους. Τα ποσοστά εξοικονόμησης και εδώ είναι ανάλογα με την εξοικονόμηση ενέργειας που είχαμε πιο πάνω. Το ενθαρρυντικό και με τα δύο αποτελέσματα είναι ότι τα ποσοστά εξοικονόμησης ενέργειας και χρόνου αυξάνονται σταδιακά όσο μεγαλώνουμε τα δεδομένα εισόδου.

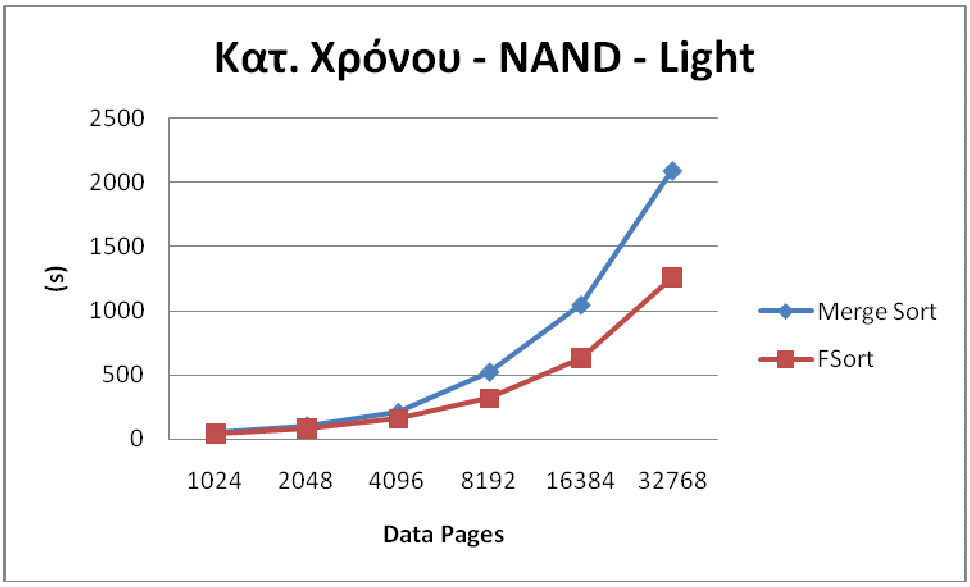
Ο Πίνακας 5 παρουσιάζει τα αποτελέσματα εκτέλεσης του ίδιου σεναρίου αλλά χρησιμοποιώντας flash τύπου NOR. Και εδώ βλέπουμε τα ίδια αποτελέσματα όπως και πιο πάνω. Αυτό είναι πολύ καλό γιατί υπάρχουν αρκετά WSDs που διαθέτουν flash τύπου NOR όπως π.χ. η οικογένεια WSDs MICA.

Ο FSort είναι πιο αποδοτικός από τον External Merge Sort σε θέματα ενέργειας και χρόνου. Επίσης από τα πειράματα είδαμε ότι γίνονται πιο λίγα erases και άρα επεκτείνεται η διάρκεια ζωής της μνήμης flash αφού τα blocks γίνονται αναξιόπιστα πιο αργά σε σχέση με τον External Merge Sort.

Πίνακας 4 Αποτελέσματα για NAND-Based Flash

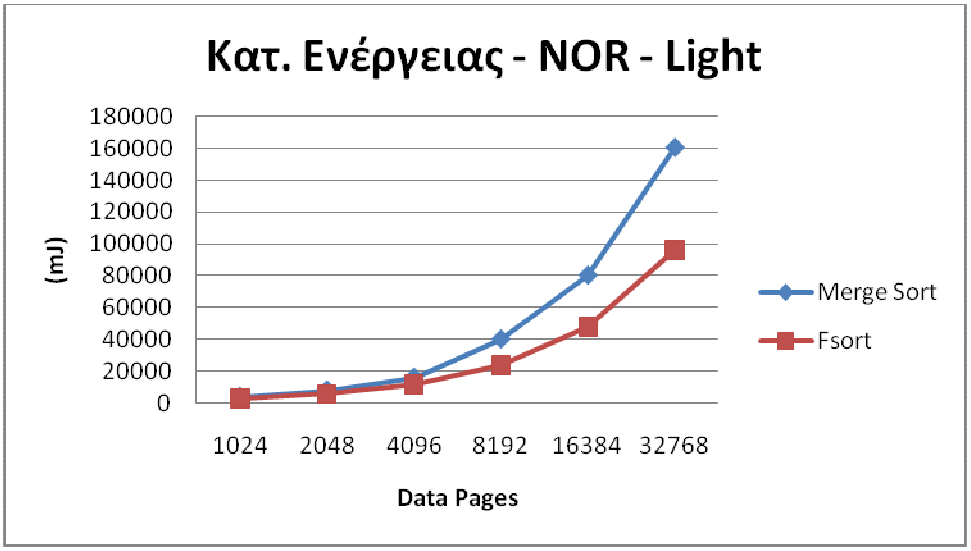
Power	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
Merge Sort	3441	6882	13765	34412	68823	137646
FSort	2581	5162	10323	20647	41294	82588
Time						
Merge Sort	52	105	209	524	1047	2094
FSort	39	79	157	314	628	1257

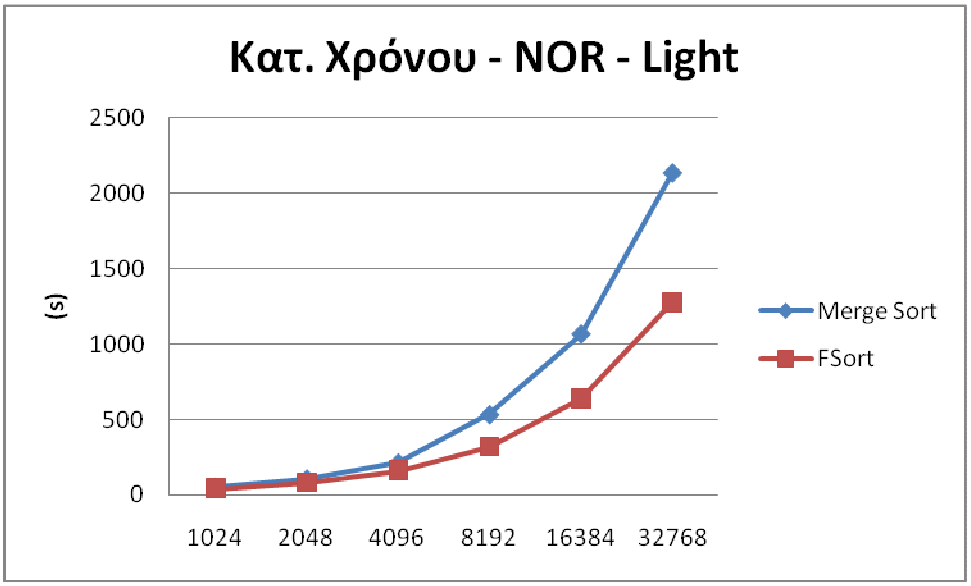




Πίνακας 5 Αποτελέσματα για NOR-Based Flash

Power	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
Merge Sort	4018	8036	16073	40182	80364	160727
Fsort	3014	6027	12055	24109	48218	96436
Time						
Merge Sort	53	106	213	532	1065	2130
FSort	40	80	160	319	639	1278


Κατ. Ενέργειας - NOR - Light
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
(mJ)
1024
2048
4096
8192
16384
32768
Data Pages
Merge Sort
Fsort


Κατ. Χρόνου - NOR - Light
2500
2000
1500
1000
500
0
(s)
1024
2048
4096
8192
16384
32768
Merge Sort
FSort

Κεφάλαιο 6

Επίλογος

6.1 Συμπεράσματα

Σε πολλές περιπτώσεις υπάρχει η ανάγκη τα δεδομένα σε WSD να είναι ταξινομημένα για να γίνονται πιο αποδοτικά κάποιες λειτουργίες όπως π.χ. οι πράξεις JOIN, επερωτήματα ισότητας (equality queries) και επερωτήματα διαστήματος (range queries). Σε αυτή την ΑΔΕ συγκρίναμε δύο προσεγγίσεις για ταξινόμηση. Αυτή του Online Sorting και Offline Sorting. Όπως αποδείξαμε και θεωρητικά αλλά και πειραματικά το Offline Sorting είναι κατά πολύ καταλληλότερο για WSD με μνήμη flash.

Σε αυτήν την ΑΔΕ, επίσης προτείναμε ένα αποδοτικό αλγόριθμο εξωτερικής ταξινόμησης για μνήμη τύπου Flash, τον FSort. Ο FSort, λαμβάνοντας υπόψη τους περιορισμούς της μνήμης Flash καθώς και τους περιορισμούς των WSDs, βελτιώνει την επίδοση από πλευράς εξοικονόμησης ενέργειας, μείωσης του χρόνου εκτέλεσης και απόκρισης. Επίσης, παρατείνει την διάρκειας ζωής της μνήμης flash.
Τα πειράματα που κάναμε έδειξαν ότι ο αλγόριθμος μειώνει τον αριθμό των read, writes και erases σε σχέση με τον External Merge Sort που χρησιμοποιήθηκε ως βάση σύγκρισης.

6.2 Μελλοντικές Εργασίες

Στο μέλλον για να έχουμε πιο ολοκληρωμένη ανάλυση των αλγορίθμων θα πρέπει να υπολογίσουμε και το κόστος ενέργειας και χρόνου της επεξεργασίας των αλγόριθμων. Επίσης θα πρέπει να γίνει σύγκριση του και με άλλους αλγόριθμους εκτός από τον External Merge Sort.

Όλα τα πειράματα που έγιναν μέχρι τώρα έγιναν κάτω από τον προσομοιωτή της μνήμης Flash. Θα ήταν σωστό να γίνουν και πειράματα σε πραγματικές συνθήκες με πραγματικές μνήμες Flash ώστε να έχουμε πιο ακριβή αποτελέσματα. Τέλος θα πρέπει να διεξαχθούν πειράματα και με άλλους τύπους flash όπως τα SSDs.
Επίσης έχουμε σχεδιάσει να βελτιώσουμε τον αλγόριθμο FSort με indexes για να έχουμε πιο αποδοτική πρόσβαση στα δεδομένα. Αυτό δεν έγινε στα πλαίσια αυτής της ΑΔΕ λόγω έλλειψης χρόνου αλλά είναι προγραμματισμένο να γίνει στο άμεσο μέλλον.

Βιβλιογραφία

[1] **Intanagonwiwat C., Govindan R. Estrin D.** Directed diffusion: A scalable and robust communication paradigm for sensor networks. *ACM MOBICOM.* 2000.

[2] **Madden S.R., Franklin M.J., Hellerstein J.M., Hong W.** TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *USENIX OSDI.* 2002.

[3] **Szewczyk R., Mainwaring A., Polastre J., Anderson J., Culler D.** An Analysis of a Large Scale Habitat Monitoring. *ACM SenSys.* 2004.

[4] *ZigBee Specification, 053474r06, Version 1.0.* 2004.

[5] *Crossbow Technology Inc.,.* [http://www.xbow.com/]

[6] **Sadler C., Zhang P., Martonosi M., Lyon S.** Hardware Design Experiences in ZebraNe. *ACM SenSys.* 2004.

[7] **Ning X., Sumit R., Krishna K.C., Deepak G., Alan B., Ramesh G., Deborah E.** A wireless sensor network for structural monitoring. *ACM SenSys.* 2004.

[8] **Zeinalipour-Yazti D., Lin S., Kalogeraki V., Gunopulos D., Najjar W.** MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. *Usenix FAST.* 2005.

[9] **Kim, Jin, Yoon, Song και Woo, Nam.** *Nonvolatile Memory and Apparatus and Method for Deciding Data Validity for the Same. U.S. Patent 2007/0189107* 16 Aug 2007.

[10] **Ban, A.** *Flash file system. U.S. Patent 5,404,485* 4 Apr 1995.

[11] **Rosenblum, Mendel and Ousterhout, John K.** The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems.* 1992.

[12] **Lee, Sang-Won και Moon, Bongki.** Design of Flash-Based DBMS: An In-Page Logging. *Proc. SIGMOD.* 2007.

[13] **D.E., Knuth.** *The Art of Computer Programming: Sorting and Searching.* s.l. : Addison Wesley, April 1997. ISBN:0-201-89685-0.

[14] **Parka H., Shim K.** FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software.* 2009.

[15] **Zeinalipour, Demetris.** Σημειώσεις μαθήματος ΕΠΛ446. 2009.

Παράρτημα Α

```
/*
 * File:     flash_emulator.c
 * Author:   Orestis Spanos
 * Desc:     This is the header file of flash_emulator.h
*/
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char uint8_t;
typedef char int8_t;

typedef unsigned short uint16_t;
typedef short int16_t;

typedef unsigned int uint32_t;
typedef int int32_t;

typedef enum{FALSE=0,TRUE=1} bool;

typedef struct
{
    void* Data;             // points to the actual data
    FILE* FilePtr;          // FILE pointer to the file contains the emulated
                            // flash memory contains in the disk
    uint16_t Blocks;        // Number of Blocks
    uint16_t PagePerBlock;  // Page per Block
    uint16_t BytesPerPage;  // Bytes Per Block
}Flash_Memory;


bool flash_create(Flash_Memory* newFlashMem,char* filename,uint16_t Blocks,uint16_t
PagePerBlock,uint16_t BytesPerPage);
bool flash_open(Flash_Memory* FlashMem,char* filename);
bool flash_close(Flash_Memory flashMemory);
bool flash_flush(Flash_Memory FlashMem);

bool flash_read(void* buffer,uint32_t len,uint32_t offset,Flash_Memory FlashMem);
bool flash_readPage(void* buffer,uint32_t pageIndex,Flash_Memory FlashMem);

bool flash_write(void* buffer,uint32_t len,uint32_t offset,Flash_Memory FlashMem);
bool flash_writePage(void* buffer,uint32_t pageIndex,Flash_Memory FlashMem);

bool flash_erase(uint32_t offset,uint32_t len,Flash_Memory FlashMem);
bool flash_eraseBlock(uint32_t blockIndex,Flash_Memory FlashMem);
bool flash_erasePage(uint32_t pageIndex,Flash_Memory FlashMem);

bool flash_import(Flash_Memory FlashMem,uint32_t offset,char *sourceFilename);

bool flash_export(Flash_Memory FlashMem,uint32_t offset,uint32_t len,char*
targetFilename);
bool flash_exportApp(Flash_Memory FlashMem,uint32_t offset,uint32_t len,char*
targetFilename);
```

```c
/*
 * File:     flash_emulator.c
 * Author:   Orestis Spanos
 * Desc:     This is the implementation of flash_emulator.h
*/
#include "flash_emulator.h"

/*
 * Creates a new virtual flash with  Blocks, PagePerBlock, BytesPerPage
 * This new virtual flash is saved in file <filename> and the
 * newFlashMem is a pointer to the new Flash Memory
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_create(Flash_Memory* newFlashMem,char* filename,uint16_t Blocks,uint16_t
PagePerBlock,uint16_t BytesPerPage)
{
    newFlashMem->Blocks=Blocks;
    newFlashMem->PagePerBlock=PagePerBlock;
    newFlashMem->BytesPerPage=BytesPerPage;
    if (!(newFlashMem->Data=malloc(Blocks*PagePerBlock*BytesPerPage))) return FALSE;
    if (!(newFlashMem->FilePtr=fopen(filename,"w+b")))
    {
        free(newFlashMem->Data);
        return FALSE;
    }

    //Write the header of the file
    fwrite(&Blocks,sizeof(uint16_t),1,newFlashMem->FilePtr);
    fwrite(&PagePerBlock,sizeof(uint16_t),1,newFlashMem->FilePtr);
    fwrite(&BytesPerPage,sizeof(uint16_t),1,newFlashMem->FilePtr);
    return TRUE;
}

/*
 * Opens a virtual flash which is saved in the file <filename> and
 * FlashMem is a pointer to this flash
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_open(Flash_Memory* FlashMem,char* filename)
{
    if (!(FlashMem->FilePtr=fopen(filename,"r+b"))) return FALSE;
    fread(&(FlashMem->Blocks),sizeof(uint16_t),1,FlashMem->FilePtr);
    fread(&(FlashMem->PagePerBlock),sizeof(uint16_t),1,FlashMem->FilePtr);
    fread(&(FlashMem->BytesPerPage),sizeof(uint16_t),1,FlashMem->FilePtr);
    if (!(FlashMem->Data=malloc(FlashMem->Blocks*FlashMem->PagePerBlock*FlashMem-
>BytesPerPage)))
    {
        fclose(FlashMem->FilePtr);
        return FALSE;
    }
    fread(FlashMem->Data,FlashMem->Blocks*FlashMem->PagePerBlock*FlashMem-
>BytesPerPage,1,FlashMem->FilePtr);
    return TRUE;
}

/*
 * Save the flash contents of FlashMem in the respective file
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_flush(Flash_Memory FlashMem)
{
    if (!FlashMem.FilePtr) return FALSE;
    fseek(FlashMem.FilePtr,3*sizeof(uint32_t),SEEK_SET);

fwrite(FlashMem.Data,FlashMem.Blocks*FlashMem.PagePerBlock*FlashMem.BytesPerPage,1,Flash
Mem.FilePtr);
    return TRUE;
}
```

```
/*
 * Save the flash contents of FlashMem in the respective file and close
 * the virtual flash. Also release all the respective resources.
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_close(Flash_Memory FlashMem)
{
    if (!(flash_flush(FlashMem))) return FALSE;
    fclose(FlashMem.FilePtr);
    free(FlashMem.Data);
    return TRUE;
}

/*
 * Read the page with index <pageIndex> of the virtual flash <FlashMem>
 * and place the contents in the <buffer>
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_readPage(void* buffer,uint32_t pageIndex,Flash_Memory FlashMem)
{

return(flash_read(buffer,FlashMem.BytesPerPage,pageIndex*FlashMem.BytesPerPage,FlashMem)
);
}

/*
 * Writes the contents of the <buffer> in the page with index <pageIndex>
 * of the virtual flash <FlashMem>
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_writePage(void* buffer,uint32_t pageIndex,Flash_Memory FlashMem)
{

return(flash_write(buffer,FlashMem.BytesPerPage,pageIndex*FlashMem.BytesPerPage,FlashMem
));
}

/*
 * Erase the block of the virtual flash memory <FlashMem> with index <blockIndex>
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_eraseBlock(uint32_t blockIndex,Flash_Memory FlashMem)
{
    uint32_t BytesPerBlock=FlashMem.PagePerBlock*FlashMem.BytesPerPage;
    return(flash_erase(blockIndex*BytesPerBlock,BytesPerBlock,FlashMem));
}

/*
 * Erase the page of the virtual flash memory <FlashMem> with index <blockIndex>
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_erasePage(uint32_t pageIndex,Flash_Memory FlashMem)
{
    return(flash_erase(pageIndex*FlashMem.BytesPerPage,FlashMem.BytesPerPage,FlashMem));
}

/*
 *
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_import(Flash_Memory FlashMem,uint32_t offset,char *sourceFilename)
{
    FILE* inp_fp=fopen(sourceFilename,"rb");
    uint32_t size;
    uint32_t flashSize=FlashMem.Blocks*FlashMem.PagePerBlock*FlashMem.BytesPerPage;
    if (!inp_fp) return FALSE;
    fseek(inp_fp,0,SEEK_END);
    size=ftell(inp_fp);
    fseek(inp_fp,0,SEEK_SET);
```

```
    if (flashSize-offset<=size) return FALSE;
    fread(FlashMem.Data+offset,size,1,inp_fp);
    return TRUE;
}

/*
 *
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_export(Flash_Memory FlashMem,uint32_t offset,uint32_t len,char*
targetFilename)
{
    FILE* outp_fp=fopen(targetFilename,"wb");
    if (!outp_fp) return FALSE;
    fwrite(FlashMem.Data+offset,len,1,outp_fp);
    fclose(outp_fp);
    return TRUE;
}

/*
 *
 *
 * returns TRUE on success, FALSE on failure
*/
bool flash_exportApp(Flash_Memory FlashMem,uint32_t offset,uint32_t len,char*
targetFilename)
{
    FILE* outp_fp=fopen(targetFilename,"r+b");
    if (!outp_fp) return FALSE;
    fseek(outp_fp,0,SEEK_END);
    fwrite(FlashMem.Data+offset,len,1,outp_fp);
    fclose(outp_fp);
    return TRUE;
}


/*
 * AUX FUNCTIONS
 */

bool flash_read(void* buffer,uint32_t len,uint32_t offset,Flash_Memory FlashMem)
{
    uint32_t flashSize=FlashMem.Blocks*FlashMem.PagePerBlock*FlashMem.BytesPerPage;
    uint32_t length=offset+len>=flashSize?flashSize-offset:len;

    if (offset>=FlashMem.Blocks*FlashMem.PagePerBlock*FlashMem.BytesPerPage) return
FALSE;
    memcpy(buffer,FlashMem.Data+offset,length);
    return TRUE;
}

bool flash_write(void* buffer,uint32_t len,uint32_t offset,Flash_Memory FlashMem)
{
    uint32_t flashSize=FlashMem.Blocks*FlashMem.PagePerBlock*FlashMem.BytesPerPage;
    uint32_t length=offset+len>=flashSize?flashSize-offset:len;

    if (offset>=flashSize) return FALSE;
    memcpy(FlashMem.Data+offset,buffer,length);
    return TRUE;
}

bool flash_erase(uint32_t offset,uint32_t len,Flash_Memory FlashMem)
{
    uint32_t flashSize=FlashMem.Blocks*FlashMem.PagePerBlock*FlashMem.BytesPerPage;
    uint32_t length=offset+len>=flashSize?flashSize-offset:len;
    if (offset>=flashSize) return FALSE;
    memset(FlashMem.Data+offset,0,length);
    return TRUE;
}
```

```c
/*
 * File:     External_Sort.h
 * Author:   Orestis Spanos
 * Desc:     This is the header file of External_Sort.h
*/
#include <stdio.h>
#include <stdlib.h>

/*
 * The struct of the records to sort
 */
typedef struct
{
    unsigned short ts;
    short val;
}RECORD;

uint16_t RecordsPerPage;

uint32_t getFileSize(char* filename)
{
    FILE *temp;
    uint32_t size;
    if (!(temp=fopen(filename,"rb"))) return 0;
    fseek(temp,0,SEEK_END);
    size=ftell(temp);
    fclose(temp);
    return size;
}
```

```
/*
 * File:     flash_emulator.c
 * Author:   Orestis Spanos
 * Desc:     This is the implementation of the Insertin Sort
*/

#include "../flash_emulator.c"
#include "../ExternalSort.h"

uint32_t inpStartingPage=0;

uint32_t totalRECORDS;
uint32_t TotalPages;
uint32_t Pages;

Flash_Memory fm;

RECORD* RamBuffer;

//For the Power/Time
uint16_t* ReadPageCounter;
uint16_t* WritePageCounter;
uint16_t* ErasePageCounter;
uint16_t* EraseBlockCounter;

double ReadPowerCons;
double WritePowerCons;
double ErasePagePowerCons;
double EraseBlockPowerCons;

double ReadTimeCons;
double WriteTimeCons;
double ErasePageTimeCons;
double EraseBlockTimeCons;

bool pageErasePolicy=TRUE;

void readPage(RECORD* bfr,uint32_t pageIndex,Flash_Memory fm)
{
    ReadPageCounter[pageIndex]++;
    flash_readPage(bfr,pageIndex,fm);
}

void writePage(RECORD* bfr,uint32_t pageIndex,Flash_Memory fm)
{
    WritePageCounter[pageIndex]++;
    flash_writePage(bfr,pageIndex,fm);
}

void eraseBlock(uint32_t blockIndex,Flash_Memory fm)
{
    EraseBlockCounter[blockIndex]++;
    flash_eraseBlock(blockIndex,fm);
}

void erasePage(uint32_t pageIndex,Flash_Memory fm)
{
    ErasePageCounter[pageIndex]++;
    flash_erasePage(pageIndex,fm);
}

void updatePage(RECORD* bfr,uint32_t pageIndex,Flash_Memory fm)
{
    uint32_t i;
    int currentBlock=pageIndex/fm.PagePerBlock;

    if (pageErasePolicy)
    {
        //ERASE PAGE
        erasePage(pageIndex,fm);
        //WRITE PAGE
        writePage(bfr,pageIndex,fm);
```

```
    }
    else
    {
        //READ BLOCK
        for (i=currentBlock*fm.PagePerBlock;i<(currentBlock+1)*fm.PagePerBlock;i++)
            if (i!=pageIndex)
                readPage(&RamBuffer[i-currentBlock*fm.PagePerBlock],i,fm);
        //UPDATE VALUE IN BUFFER
        RamBuffer[pageIndex%fm.PagePerBlock]=*bfr;
        //ERASE BLOCK
        eraseBlock(currentBlock,fm);
        //WRITE BACK IN BLOCK
        for (i=currentBlock*fm.PagePerBlock;i<(currentBlock+1)*fm.PagePerBlock;i++)
            writePage(&RamBuffer[i-currentBlock*fm.PagePerBlock],i,fm);
    }
}
void calculateResults()
{
    int i;
    int totalReads=0,totalWrites=0,totalErases=0,totalBlock=0;
    double totalPower=0,totalTime=0;
    char dump[64];
    char resultFilename[64];

    FILE* out_fp;
    FILE* cfg_fp=fopen("configuration.txt","r");
    fscanf(cfg_fp,"%s %lf",dump,&ReadPowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&WritePowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&ErasePagePowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&EraseBlockPowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&ReadTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&WriteTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&ErasePageTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&EraseBlockTimeCons);
    fscanf(cfg_fp,"%s %s",dump,resultFilename);
    out_fp=fopen(resultFilename,"w");
    fprintf(out_fp,"");
    fprintf(out_fp,"Page");

    for (i=0;i<Pages;i++)
    {
        totalReads+=ReadPageCounter[i];
        totalWrites+=WritePageCounter[i];
        totalErases+=ErasePageCounter[i];
        if (i<fm.Blocks)
        {
            totalBlock+=EraseBlockCounter[i];

fprintf(out_fp,"%d%d%d%d%d",i,ReadPageCounter[i],WritePageCounter[i],ErasePageCounter[i]
,EraseBlockCounter[i]);
        }
        else
fprintf(out_fp,"%d%d%d%d",i,ReadPageCounter[i],WritePageCounter[i],ErasePageCounter[i]);
    }
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"%d%d%d%d",totalReads,totalWrites,totalErases,totalBlock);
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"Power Analysis");
    fprintf(out_fp,"Read:%lf",totalReads*ReadPowerCons);
    fprintf(out_fp,"Write:%lf",totalWrites*WritePowerCons);
    if (pageErasePolicy) fprintf(out_fp,"Erase
Page:%lf",totalErases*ErasePagePowerCons);
    else fprintf(out_fp,"Erase Block:%lf",totalBlock*EraseBlockPowerCons);

totalPower=totalReads*ReadPowerCons+totalWrites*WritePowerCons+totalErases*ErasePagePowe
rCons+totalBlock*EraseBlockPowerCons;
    fprintf(out_fp,"TOTAL:%lf",totalPower);
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"Time Analysis");
    fprintf(out_fp,"Read:%lf",totalReads*ReadTimeCons);
    fprintf(out_fp,"Write:%lf",totalWrites*WriteTimeCons);
    if (pageErasePolicy) fprintf(out_fp,"Erase Page:%lf",totalErases*ErasePageTimeCons);
    else fprintf(out_fp,"Erase Block:%lf",totalBlock*EraseBlockTimeCons);
```

```
totalTime=totalReads*ReadTimeCons+totalWrites*WriteTimeCons+totalErases*ErasePageTimeCon
s+totalBlock*EraseBlockTimeCons;
    fprintf(out_fp,"TOTAL:%lf",totalTime);
}

int main()
{
    uint32_t i,j;
    RECORD index;
    RECORD temp;
    flash_create(&fm,"AT45DB041B.efm",16896,8,4);
    flash_import(fm,inpStartingPage*fm.BytesPerPage,"input.txt");
    RamBuffer=(RECORD *)malloc(fm.PagePerBlock*fm.BytesPerPage);

    RecordsPerPage=fm.BytesPerPage/sizeof(RECORD);
    totalRECORDS=getFileSize("input.txt")/sizeof(RECORD);
    TotalPages=totalRECORDS/RecordsPerPage;

    Pages=fm.Blocks*fm.PagePerBlock;

    ReadPageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
    WritePageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
    ErasePageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
    EraseBlockCounter=(uint16_t*)malloc(fm.Blocks*sizeof(uint16_t));

    for (i=0;i<Pages;i++)
    {
        ReadPageCounter[i]=0;
        WritePageCounter[i]=0;
        ErasePageCounter[i]=0;
    }

    for (i=0;i<fm.Blocks;i++) EraseBlockCounter[i]=0;

    for (i=1;i<totalRECORDS;i++)
    {
        readPage(&index,i,fm);
        readPage(&temp,i-1,fm);
        j = i;
        while ((j > 0) && (temp.val > index.val))
        {
            updatePage(&temp,j,fm);
            j = j - 1;
            readPage(&temp,j-1,fm);
        }
        updatePage(&index,j,fm);
    }

    flash_export(fm,inpStartingPage*fm.BytesPerPage,totalRECORDS*sizeof(RECORD),"out");

    flash_close(fm);

    calculateResults();
    system("pause");
}
```

```
/*
 * File:     flash_emulator.c
 * Author:   Orestis Spanos
 * Desc:     This is the implementation of N-Way Merge Sort
*/

#include <math.h>
#include "../flash_emulator.c"
#include "../ExternalSort.h"

uint32_t inpStartingPage=0;
uint32_t outpStartingPage;

uint32_t totalRECORDS;

uint16_t RecordsPerPage;

uint32_t Pages;

uint16_t TotalPages;
uint16_t RecordsInLastPage;

Flash_Memory fm;

char* configFile="configuration.txt";
//For the mergesort algorithm
uint8_t BufferPages=9;
uint8_t NWay=8;
RECORD* ramBuffer;
RECORD** ReadBuffer;
RECORD* WriteBuffer;

RECORD **RBPtr;
uint16_t *RBCount;
uint16_t *RBLastPageReaded;
uint16_t WBCount;


//For the Power/Time
uint16_t* ReadPageCounter;
uint16_t* WritePageCounter;
uint16_t* ErasePageCounter;
uint16_t* EraseBlockCounter;

double ReadPowerCons;
double WritePowerCons;
double ErasePagePowerCons;
double EraseBlockPowerCons;

double ReadTimeCons;
double WriteTimeCons;
double ErasePageTimeCons;
double EraseBlockTimeCons;

bool pageErasePolicy=FALSE;

void shellsort(RECORD a[],uint32_t n)
{
    int32_t j,i,m;
    RECORD mid;
    for (m = n/2;m>0;m/=2)
    {
        for (j = m;j< n;j++)
        {
            for (i=j-m;i>=0;i-=m)
            {
                if (a[i+m].val>=a[i].val)
                    break;
                else
                {
                    mid = a[i];
                    a[i] = a[i+m];
```

```
                    a[i+m] = mid;
                }
            }
        }
    }
}

unsigned char notNull(RECORD* a[])
{
    unsigned short i;
    for (i=0;i<NWay;i++) if (a[i]!=NULL) return 1;
    return 0;
}

unsigned short min(RECORD* RBPtr[])
{
    unsigned short ret=0;
    unsigned short min;

    while (RBPtr[ret]==NULL) ret++;
    min=ret;
    ret++;
    while (ret<NWay)
    {
        if (RBPtr[ret]!=NULL && RBPtr[ret]->val<RBPtr[min]->val) min=ret;
        ret++;
    }

    return min;
}

void readPage(RECORD* buffer,uint32_t start,uint32_t pageIndex)
{
    ReadPageCounter[(start+pageIndex)%Pages]++;
    flash_readPage(buffer,(start+pageIndex)%Pages,fm);
}

void readPages(RECORD* buffer,uint32_t start,uint32_t pageIndex,uint8_t count)
{
    int i;
    for (i=0;i<count;i++) readPage(&buffer[i*RecordsPerPage],start,pageIndex+i);
}

void writePage(RECORD* buffer,uint32_t start,uint32_t pageIndex)
{
    if (pageErasePolicy)
    {
        ErasePageCounter[(start+pageIndex)%Pages]++;
        flash_erasePage((start+pageIndex)%Pages,fm);
    }
    else
    {
        if ((start+pageIndex)%fm.PagePerBlock==0)
        {
            EraseBlockCounter[((start+pageIndex)%Pages)/fm.PagePerBlock]++;
            flash_eraseBlock(((start+pageIndex)%Pages)/fm.PagePerBlock,fm);
        }
    }
    WritePageCounter[(start+pageIndex)%Pages]++;
    flash_writePage(buffer,(start+pageIndex)%Pages,fm);
}

void writePages(RECORD* buffer,uint32_t start,uint32_t pageIndex,uint8_t count)
{
    int i;
    for (i=0;i<count;i++) writePage(&buffer[i*RecordsPerPage],start,pageIndex+i);
}

void export(uint32_t fp,uint32_t countRECORDS,char* filename)
{
    if (fp*fm.BytesPerPage+countRECORDS*sizeof(RECORD)>Pages*fm.BytesPerPage)
    {
        //SPLIT
        flash_export(fm,fp*fm.BytesPerPage,((Pages-
fp)*RecordsPerPage)*sizeof(RECORD),filename);
```

```
        flash_exportApp(fm,0,((countRECORDS-(Pages-
fp)*RecordsPerPage))*sizeof(RECORD),filename);
    }
    else
    {
        flash_export(fm,fp*fm.BytesPerPage,countRECORDS*sizeof(RECORD),filename);
    }
}

void calculateResults()
{
    int i;
    int totalReads=0,totalWrites=0,totalErases=0,totalBlock=0;
    double totalPower=0,totalTime=0;
    char dump[64];
    char resultFilename[64];
    FILE* out_fp;
    FILE* cfg_fp=fopen(configFile,"r");
    fscanf(cfg_fp,"%s %lf",dump,&ReadPowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&WritePowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&ErasePagePowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&EraseBlockPowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&ReadTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&WriteTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&ErasePageTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&EraseBlockTimeCons);
    fscanf(cfg_fp,"%s %s",dump,resultFilename);
    out_fp=fopen(resultFilename,"w");
    fprintf(out_fp,"");
    fprintf(out_fp,"Page");
    for (i=0;i<Pages;i++)
    {
        totalReads+=ReadPageCounter[i];
        totalWrites+=WritePageCounter[i];
        totalErases+=ErasePageCounter[i];
        if (i<fm.Blocks)
        {
            totalBlock+=EraseBlockCounter[i];

fprintf(out_fp,"%d%d%d%d%d",i,ReadPageCounter[i],WritePageCounter[i],ErasePageCounter[i]
,EraseBlockCounter[i]);
        }
        else
fprintf(out_fp,"%d%d%d%d",i,ReadPageCounter[i],WritePageCounter[i],ErasePageCounter[i]);
    }
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"%d%d%d%d",totalReads,totalWrites,totalErases,totalBlock);
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"Power Analysis");
    fprintf(out_fp,"Read:%lf",totalReads*ReadPowerCons);
    fprintf(out_fp,"Write:%lf",totalWrites*WritePowerCons);
    if (pageErasePolicy) fprintf(out_fp,"Erase
Page:%lf",totalErases*ErasePagePowerCons);
    else fprintf(out_fp,"Erase Block:%lf",totalBlock*EraseBlockPowerCons);

totalPower=totalReads*ReadPowerCons+totalWrites*WritePowerCons+totalErases*ErasePagePowe
rCons+totalBlock*EraseBlockPowerCons;
    fprintf(out_fp,"TOTAL:%lf",totalPower);
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"Time Analysis");
    fprintf(out_fp,"Read:%lf",totalReads*ReadTimeCons);
    fprintf(out_fp,"Write:%lf",totalWrites*WriteTimeCons);
    if (pageErasePolicy) fprintf(out_fp,"Erase Page:%lf",totalErases*ErasePageTimeCons);
    else fprintf(out_fp,"Erase Block:%lf",totalBlock*EraseBlockTimeCons);

totalTime=totalReads*ReadTimeCons+totalWrites*WriteTimeCons+totalErases*ErasePageTimeCon
s+totalBlock*EraseBlockTimeCons;
    fprintf(out_fp,"TOTAL:%lf",totalTime);
}

int main()
{
    int i,j,k;
    int countRECORDS=0;
```

```
unsigned int page,sortedRuns;

flash_create(&fm,"AT45DB041B.efm",16896,8,4);
flash_import(fm,inpStartingPage*fm.BytesPerPage,"input.txt");

RecordsPerPage=fm.BytesPerPage/sizeof(RECORD);
totalRECORDS=getFileSize("input.txt")/sizeof(RECORD);
TotalPages=totalRECORDS/RecordsPerPage;
RecordsInLastPage=totalRECORDS%RecordsPerPage;
Pages=fm.Blocks*fm.PagePerBlock;
if (RecordsInLastPage==0) RecordsInLastPage=RecordsPerPage;
else TotalPages++;

outpStartingPage=(inpStartingPage+TotalPages)%Pages;

//Memory Allocation

ramBuffer=(RECORD*)malloc(BufferPages*RecordsPerPage*sizeof(RECORD));
ReadBuffer=(RECORD**)malloc(NWay*sizeof(RECORD*));
for (i=0;i<NWay;i++) ReadBuffer[i]=&ramBuffer[i*RecordsPerPage];
WriteBuffer=&ramBuffer[i*RecordsPerPage];
RBPtr=(RECORD**)malloc(NWay*sizeof(RECORD*));
RBCount=(uint16_t*)malloc(NWay*sizeof(uint16_t));
RBLastPageReaded=(uint16_t*)malloc(NWay*sizeof(uint16_t));

ReadPageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
WritePageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
ErasePageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
EraseBlockCounter=(uint16_t*)malloc(fm.Blocks*sizeof(uint16_t));

for (i=0;i<Pages;i++)
{
    ReadPageCounter[i]=0;
    WritePageCounter[i]=0;
    ErasePageCounter[i]=0;
}

for (i=0;i<fm.Blocks;i++) EraseBlockCounter[i]=0;

//STEP 0
for (page=0;page<TotalPages;page+=BufferPages)
{
    // READ
    countRECORDS=0;

    for (j=0,i=page;j<BufferPages&&i<TotalPages;j++,i++)
    {
        readPage(&ramBuffer[j*RecordsPerPage],inpStartingPage,i);
        if (i==TotalPages-1)countRECORDS+=RecordsInLastPage;
        else countRECORDS+=RecordsPerPage;
    }

    // SORT
    shellsort(ramBuffer,countRECORDS);

    // WRITE BACK
    for (j=0,i=page;j<BufferPages&&i<TotalPages;j++,i++)
    {
        writePage(&ramBuffer[j*RecordsPerPage],outpStartingPage,i);
    }
}
sortedRuns=BufferPages;

//STEP 1
while (sortedRuns<TotalPages)
{
    //Swap input output
    inpStartingPage=outpStartingPage;
    outpStartingPage=(inpStartingPage+TotalPages)%Pages;

    page=0;
    while (page<TotalPages)
    {
        //READ Buffers
        for (i=0,j=page;i<NWay&&j<TotalPages;i++,j+=sortedRuns)
```

```
            {
                RBPtr[i]=ReadBuffer[i];
                RBLastPageReaded[i]=j;
                if (TotalPages-j>sortedRuns)RBCount[i]=sortedRuns*RecordsPerPage;
                else RBCount[i]=totalRECORDS-(j*RecordsPerPage);
                readPage(ReadBuffer[i],inpStartingPage,j);
            }

            for (;i<NWay;i++)
            {
                RBPtr[i]=NULL;
                RBCount[i]=0;
            }

            //MERGE AND WRITE BACK
            WBCount=0;

            while (notNull(RBPtr)==1)
            {
                k=min(RBPtr);
                WriteBuffer[WBCount%RecordsPerPage]=*RBPtr[k];
                RBPtr[k]++;
                RBCount[k]--;
                WBCount++;
                if (RBCount[k]==0) RBPtr[k]=NULL;
                else if (RBPtr[k]-ReadBuffer[k]==RecordsPerPage)
                {
                    //READ
                    RBLastPageReaded[k]++;
                    readPage(ReadBuffer[k],inpStartingPage,RBLastPageReaded[k]);
                    RBPtr[k]=ReadBuffer[k];
                }

                if (WBCount%RecordsPerPage==0)
                {
                    //WRITE BACK
                    writePage(WriteBuffer,outpStartingPage,(page-
1+WBCount/RecordsPerPage));
                }
            }
            //WRITE BACK IF ANY
            if (WBCount%RecordsPerPage!=0)
            {
                writePage(WriteBuffer,outpStartingPage,(page+WBCount/RecordsPerPage));
            }
            page+=NWay*sortedRuns;
        }
        sortedRuns*=NWay;
    }

    //Export the Output
    export(outpStartingPage,totalRECORDS,"out");
    calculateResults();
    flash_close(fm);
    system("pause");
}
```

```
/*
 * File:     flash_emulator.c
 * Author:   Orestis Spanos
 * Desc:     This is the implementation of FSort
*/
#include <math.h>
#include "../flash_emulator.c"
#include "../ExternalSort.h"


uint32_t inpStartingPage=0;
uint32_t outpStartingPage;

uint32_t totalRECORDS;

uint16_t RecordsPerPage;

uint32_t Pages;

uint16_t TotalPages;
uint16_t RecordsInLastPage;


Flash_Memory fm;


char* configFile="configuration.txt";
//For the mergesort algorithm
uint8_t BufferPages=9;
uint8_t NWay=8;
RECORD* ramBuffer;
RECORD** ReadBuffer;
RECORD* WriteBuffer;

RECORD **RBPtr;
uint16_t *RBCount;
uint16_t *RBLastPageReaded;
uint16_t WBCount;
uint32_t sortedRuns[2048];
uint32_t sortedRunsCount=0,newSortedRunsCount;

//For the Power/Time
uint16_t* ReadPageCounter;
uint16_t* WritePageCounter;
uint16_t* ErasePageCounter;
uint16_t* EraseBlockCounter;

double ReadPowerCons;
double WritePowerCons;
double ErasePagePowerCons;
double EraseBlockPowerCons;

double ReadTimeCons;
double WriteTimeCons;
double ErasePageTimeCons;
double EraseBlockTimeCons;

bool pageErasePolicy=FALSE;

void shellsort(RECORD a[],uint32_t n)
{
    int32_t j,i,m;
    RECORD mid;
    for (m = n/2;m>0;m/=2)
    {
        for (j = m;j< n;j++)
        {
            for (i=j-m;i>=0;i-=m)
            {
                if (a[i+m].val>=a[i].val)
                    break;
                else
```

```
                {
                    mid = a[i];
                    a[i] = a[i+m];
                    a[i+m] = mid;
                }
            }
        }
    }
}

unsigned char notNull(RECORD* a[])
{
    unsigned short i;
    for (i=0;i<NWay;i++) if (a[i]!=NULL) return 1;
    return 0;
}

unsigned short min(RECORD* RBPtr[])
{
    unsigned short ret=0;
    unsigned short min;

    while (RBPtr[ret]==NULL) ret++;
    min=ret;
    ret++;
    while (ret<NWay)
    {
        if (RBPtr[ret]!=NULL && RBPtr[ret]->val<RBPtr[min]->val) min=ret;
        ret++;
    }

    return min;
}

void readPage(RECORD* buffer,uint32_t start,uint32_t pageIndex)
{
    ReadPageCounter[(start+pageIndex)%Pages]++;
    flash_readPage(buffer,(start+pageIndex)%Pages,fm);
}

void readPages(RECORD* buffer,uint32_t start,uint32_t pageIndex,uint8_t count)
{
    int i;
    for (i=0;i<count;i++) readPage(&buffer[i*RecordsPerPage],start,pageIndex+i);
}

void writePage(RECORD* buffer,uint32_t start,uint32_t pageIndex)
{
    if (pageErasePolicy)
    {
        ErasePageCounter[(start+pageIndex)%Pages]++;
        flash_erasePage((start+pageIndex)%Pages,fm);
    }
    else
    {
        if ((start+pageIndex)%fm.PagePerBlock==0)
        {
            EraseBlockCounter[((start+pageIndex)%Pages)/fm.PagePerBlock]++;
            flash_eraseBlock(((start+pageIndex)%Pages)/fm.PagePerBlock,fm);
        }
    }
    WritePageCounter[(start+pageIndex)%Pages]++;
    flash_writePage(buffer,(start+pageIndex)%Pages,fm);
}

void writePages(RECORD* buffer,uint32_t start,uint32_t pageIndex,uint8_t count)
{
    int i;
    for (i=0;i<count;i++) writePage(&buffer[i*RecordsPerPage],start,pageIndex+i);
}

void export(uint32_t fp,uint32_t countRECORDS,char* filename)
{
    if (fp*fm.BytesPerPage+countRECORDS*sizeof(RECORD)>Pages*fm.BytesPerPage)
    {
```

```
        //SPLIT
        flash_export(fm,fp*fm.BytesPerPage,((Pages-
fp)*RecordsPerPage)*sizeof(RECORD),filename);
        flash_exportApp(fm,0,((countRECORDS-(Pages-
fp)*RecordsPerPage))*sizeof(RECORD),filename);
    }
    else
    {
        flash_export(fm,fp*fm.BytesPerPage,countRECORDS*sizeof(RECORD),filename);
    }
}

void calculateResults()
{
    int i;
    int totalReads=0,totalWrites=0,totalErases=0,totalBlock=0;
    double totalPower=0,totalTime=0;
    char dump[64];
    char resultFilename[64];
    FILE* out_fp;
    FILE* cfg_fp=fopen(configFile,"r");
    fscanf(cfg_fp,"%s %lf",dump,&ReadPowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&WritePowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&ErasePagePowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&EraseBlockPowerCons);
    fscanf(cfg_fp,"%s %lf",dump,&ReadTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&WriteTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&ErasePageTimeCons);
    fscanf(cfg_fp,"%s %lf",dump,&EraseBlockTimeCons);
    fscanf(cfg_fp,"%s %s",dump,resultFilename);
    out_fp=fopen(resultFilename,"w");
    fprintf(out_fp,"");
    fprintf(out_fp,"Page");
    for (i=0;i<Pages;i++)
    {
        totalReads+=ReadPageCounter[i];
        totalWrites+=WritePageCounter[i];
        totalErases+=ErasePageCounter[i];
        if (i<fm.Blocks)
        {
            totalBlock+=EraseBlockCounter[i];

fprintf(out_fp,"%d%d%d%d%d",i,ReadPageCounter[i],WritePageCounter[i],ErasePageCounter[i]
,EraseBlockCounter[i]);
        }
        else
fprintf(out_fp,"%d%d%d%d",i,ReadPageCounter[i],WritePageCounter[i],ErasePageCounter[i]);
    }
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"%d%d%d%d",totalReads,totalWrites,totalErases,totalBlock);
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"Power Analysis");
    fprintf(out_fp,"Read:%lf",totalReads*ReadPowerCons);
    fprintf(out_fp,"Write:%lf",totalWrites*WritePowerCons);
    if (pageErasePolicy) fprintf(out_fp,"Erase
Page:%lf",totalErases*ErasePagePowerCons);
    else fprintf(out_fp,"Erase Block:%lf",totalBlock*EraseBlockPowerCons);

totalPower=totalReads*ReadPowerCons+totalWrites*WritePowerCons+totalErases*ErasePagePowe
rCons+totalBlock*EraseBlockPowerCons;
    fprintf(out_fp,"TOTAL:%lf",totalPower);
    fprintf(out_fp,"----------------------------------------");
    fprintf(out_fp,"Time Analysis");
    fprintf(out_fp,"Read:%lf",totalReads*ReadTimeCons);
    fprintf(out_fp,"Write:%lf",totalWrites*WriteTimeCons);
    if (pageErasePolicy) fprintf(out_fp,"Erase Page:%lf",totalErases*ErasePageTimeCons);
    else fprintf(out_fp,"Erase Block:%lf",totalBlock*EraseBlockTimeCons);

totalTime=totalReads*ReadTimeCons+totalWrites*WriteTimeCons+totalErases*ErasePageTimeCon
s+totalBlock*EraseBlockTimeCons;
    fprintf(out_fp,"TOTAL:%lf",totalTime);
}

int main()
```

```
{
    int i,j,k;
    int countRECORDS=0;
    unsigned int page,sr;
    short minKey;
    flash_create(&fm,"AT45DB041B.efm",16896,8,4);
    flash_import(fm,inpStartingPage*fm.BytesPerPage,"input.txt");

    RecordsPerPage=fm.BytesPerPage/sizeof(RECORD);
    totalRECORDS=getFileSize("input.txt")/sizeof(RECORD);
    TotalPages=totalRECORDS/RecordsPerPage;
    RecordsInLastPage=totalRECORDS%RecordsPerPage;
    Pages=fm.Blocks*fm.PagePerBlock;
    if (RecordsInLastPage==0) RecordsInLastPage=RecordsPerPage;
    else TotalPages++;

    outpStartingPage=(inpStartingPage+TotalPages)%Pages;

    //Memory Allocation

    ramBuffer=(RECORD*)malloc(BufferPages*RecordsPerPage*sizeof(RECORD));
    ReadBuffer=(RECORD**)malloc(NWay*sizeof(RECORD*));
    for (i=0;i<NWay;i++) ReadBuffer[i]=&ramBuffer[i*RecordsPerPage];
    WriteBuffer=&ramBuffer[i*RecordsPerPage];
    RBPtr=(RECORD**)malloc(NWay*sizeof(RECORD*));
    RBCount=(uint16_t*)malloc(NWay*sizeof(uint16_t));
    RBLastPageReaded=(uint16_t*)malloc(NWay*sizeof(uint16_t));

    ReadPageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
    WritePageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
    ErasePageCounter=(uint16_t*)malloc(Pages*sizeof(uint16_t));
    EraseBlockCounter=(uint16_t*)malloc(fm.Blocks*sizeof(uint16_t));

    for (i=0;i<Pages;i++)
    {
        ReadPageCounter[i]=0;
        WritePageCounter[i]=0;
        ErasePageCounter[i]=0;
    }

    for (i=0;i<fm.Blocks;i++) EraseBlockCounter[i]=0;
    /*
        for (i=0;i<TotalPages-1;i++)
        {
            flash_readPage(ramBuffer,(inpStartingPage+i)%Pages,fm);
            shellsort(ramBuffer,RecordsPerPage);
            flash_writePage(ramBuffer,(inpStartingPage+i)%Pages,fm);
        }
        flash_readPage(ramBuffer,(inpStartingPage+i)%Pages,fm);
        shellsort(ramBuffer,RecordsInLastPage);
        flash_writePage(ramBuffer,(inpStartingPage+i)%Pages,fm);*/
    ////////////////////////////////////////////////////////////////////////////
    //OUR ALGORITHM
    //Create sorted runs
    countRECORDS=0;
    sortedRuns[0]=0;
    for (i=0;i<NWay&&i<TotalPages;i++)
    {
        readPage(ReadBuffer[i],inpStartingPage,i);
        RBLastPageReaded[i]=i;
        RBPtr[i]=ReadBuffer[i];
        if (i==TotalPages-1) RBCount[i]=RecordsInLastPage;
        else RBCount[i]=RecordsPerPage;
    }

    for (;i<NWay;i++)
    {
        RBPtr[i]=NULL;
        RBCount[i]=0;
    }
    WBCount=0;
    while (countRECORDS<totalRECORDS)
    {
        k=min(RBPtr);
        minKey=RBPtr[k]->val;
```

```
        WriteBuffer[WBCount%RecordsPerPage]=*RBPtr[k];
        RBPtr[k]++;
        RBCount[k]--;
        WBCount++;
        countRECORDS++;

        if (RBCount[k]==0)
        {
            if (RBLastPageReaded[k]+NWay<TotalPages)
            {
                //Read a new page
                readPage(ReadBuffer[k],inpStartingPage,RBLastPageReaded[k]+NWay);
                RBLastPageReaded[k]+=NWay;
                RBPtr[k]=ReadBuffer[k];
                if (RBLastPageReaded[k]==TotalPages-1) RBCount[k]=RecordsInLastPage;
                else RBCount[k]=RecordsPerPage;
            }
            else RBPtr[k]=NULL;

        }

        if (RBPtr[k] && minKey>RBPtr[k]->val) RBPtr[k]=NULL;

        if (WBCount%RecordsPerPage==0)
        {
            //WRITE BACK
            writePage(WriteBuffer,outpStartingPage,(WBCount/RecordsPerPage)-1);
        }

        if (notNull(RBPtr)!=1)
        {
            //END SORTED
            sortedRunsCount++;
            sortedRuns[sortedRunsCount]=countRECORDS;

            for (i=0;i<NWay;i++)
            {
                if (RBCount[i]==0)
                {
                    if (RBLastPageReaded[i]+NWay<TotalPages)
                    {
                        //Read a new page

readPage(ReadBuffer[i],inpStartingPage,RBLastPageReaded[i]+NWay);
                        RBLastPageReaded[i]+=NWay;
                        RBPtr[i]=ReadBuffer[i];
                        if (RBLastPageReaded[i]==TotalPages-1)
RBCount[i]=RecordsInLastPage;
                        else RBCount[i]=RecordsPerPage;
                    }
                    else RBPtr[i]=NULL;
                }
                else
                {
                    RBPtr[i]=&ReadBuffer[i][RecordsPerPage-RBCount[i]];
                }
            }
        }
    }
    //WRITE BACK IF ANY
    if (WBCount%RecordsPerPage!=0)
    {
        writePage(WriteBuffer,outpStartingPage,WBCount/RecordsPerPage);
    }
    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
    //Phase 2 Merging
    while (sortedRunsCount>1)
    {
        //Swap input output
        inpStartingPage=outpStartingPage;
        outpStartingPage=(inpStartingPage+TotalPages)%Pages;

        newSortedRunsCount=0;
        for (sr=0;sr<sortedRunsCount;sr+=NWay)
```

```
        {
            page=sortedRuns[sr];
            for (i=0,j=sr;i<NWay && j<sortedRunsCount;i++,j++)
            {
                readPage(ReadBuffer[i],inpStartingPage,sortedRuns[j]);
                RBLastPageReaded[i]=sortedRuns[j];
                RBPtr[i]=ReadBuffer[i];
                RBCount[i]=sortedRuns[j+1]-sortedRuns[j];
            }

            for (;i<NWay;i++)
            {
                RBPtr[i]=NULL;
                RBCount[i]=0;
            }

            //MERGE AND WRITE BACK
            WBCount=0;

            while (notNull(RBPtr)==1)
            {
                k=min(RBPtr);
                WriteBuffer[WBCount%RecordsPerPage]=*RBPtr[k];
                RBPtr[k]++;
                RBCount[k]--;
                WBCount++;
                if (RBCount[k]==0) RBPtr[k]=NULL;
                else if (RBPtr[k]-ReadBuffer[k]==RecordsPerPage)
                {
                    //READ
                    RBLastPageReaded[k]++;
                    readPage(ReadBuffer[k],inpStartingPage,RBLastPageReaded[k]);
                    RBPtr[k]=ReadBuffer[k];
                }

                if (WBCount%RecordsPerPage==0)
                {
                    //WRITE BACK
                    writePage(WriteBuffer,outpStartingPage,(page-
1+WBCount/RecordsPerPage));
                }
            }
            //WRITE BACK IF ANY
            if (WBCount%RecordsPerPage!=0)
            {
                writePage(WriteBuffer,outpStartingPage,(page+WBCount/RecordsPerPage));
            }
            newSortedRunsCount++;
            sortedRuns[newSortedRunsCount]=sortedRuns[newSortedRunsCount-1]+WBCount;

        }
        sortedRunsCount=newSortedRunsCount;
    }


    ////////////////////////////////////////////////////////////////////////
    //Export the Output
    export(outpStartingPage,totalRECORDS,"out");
    calculateResults();
    //calculateResults();
    flash_close(fm);
    system("pause");
}
```

FSort: External Sorting on Flash-based Sensor Devices

Panayiotis Andreou, Orestis Spanos, Demetrios Zeinalipour-Yazti,
George Samaras, Panos K. Chrysanthis[‡]

Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus
[‡] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
{panic,cs05os1,dzeina,cssamara}@cs.ucy.ac.cy, panos@cs.pitt.edu

ABSTRACT

Wireless Sensor Networks (WSNs) typically execute continuous queries that periodically record measurements from the sensors and transmit them to a base station. More than often, users post queries whose performance can benefit from the existence of sorted data (e.g., range, top-k, join queries). In long-term deployments, due to the high cost of recurring communications, it is more efficient to store an immense number of measurements locally, sort them locally and only transmit them when requested. Since the beginning of Wireless Sensor Devices (WSDs), flash memory has been the most prevailing storage medium for stockpiling and processing measurements locally. However, due to the unique characteristics of flash memory, traditional disk-based sorting algorithms, that concentrate on minimizing I/O operations, are rendered inefficient. This, along with the limited battery, processing and memory capabilities of WSDs necessitates the development of new flash-aware and power-efficient algorithms. In this paper, we present, $FSort$, an efficient external sorting algorithm for WSDs. Our $FSort$ algorithm utilizes a small footprint procedure that traverses flash memory in a manner that releases continuous blocks of pages that can be efficiently erased and reprogrammed. We experimentally validate our approach using real datasets and show that $FSort$ increases the overall performance of sorting compared to traditional approaches while in parallel decreasing the wearability of flash memory.

1. INTRODUCTION

The improvements in hardware design along with the wide availability of economically viable embedded *sensor devices* make it feasible today to interact and understand the physical world at an extremely high fidelity [22, 20, 12]. Applications of *Wireless Sensor Networks* (WSNs) devices range from environmental monitoring (such as atmosphere and habitant monitoring [22, 19, 3]) to seismic and structural [15] monitoring as well as industry manufacturing [5, 12].

In many WSNs, recorded measurements are continuously transmitted to a base station for storage and analysis. However, in long-term deployments it is often preferred to store measurements locally at each sensor and transmit them to the user only when requested. Such a scheme is particularly favored because communication over the radio in a WSN is far more energy demanding than all other functions, such as storage [25, 2, 1] and processing [12, 21, 23, 24].

Although storing the data locally is more efficient than transmitting it continuously to a sink point, the storing process by itself raises some important challenges. For instance when users perform *range queries* by a given attribute, (e.g., "Find GPS locations where humidity is in the range A to B"), then that requires that the data is sequentially organized (i.e., sorted) by the humidity attribute. Otherwise, the query will end up traversing a very large number of data pages incurring a large read cost. Notice that reading data off the flash card in large numbers or repetively, has a significant cost in its own right. Another example where data needs to be sorted on flash is when a query aims to *JOIN* data under a specific attribute.Such JOIN operations are executed more swiftly when sorted data exist [14].

Sorting in WSDs can be performed in two modes, a) *online*: the data are continuously sorted upon the acquisition of new measurements from the sensors, and b) *offline*: the data are stored on flash using some database scheme and are periodically sorted upon request. The question rises which of the two approaches delivers the best performance on flash-based WSDs. Our experimental study presented in Section 5 suggests that online sorting is two to three orders of magnitudes worse than offline sorting both for NAND and NOR flash memory.

The majority of WSDs utilizes NAND-based flash memory for storing local measurements; we will further describe this specific type of flash memory in Section 2.1. NAND-based flash memory has some unique characteristics which are listed below. Later, in Section 4, we will address these constraints when presenting the FSort algorithm.

- **Write-Constraint A:** The minimum unit of write operation in flash is a page. Page sizes typically vary from 256B to 4KB.
- **Write-Constraint B:** Pages in flash can only be written in sequential order, i.e., after page i has been written, any page $j : 1 \leq j < i$ can not be written even if it is empty.
- **Erase-Constraint:** A page i cannot be deleted unless the block that contains page i has been erased.

DMSN '09 Lyon, France

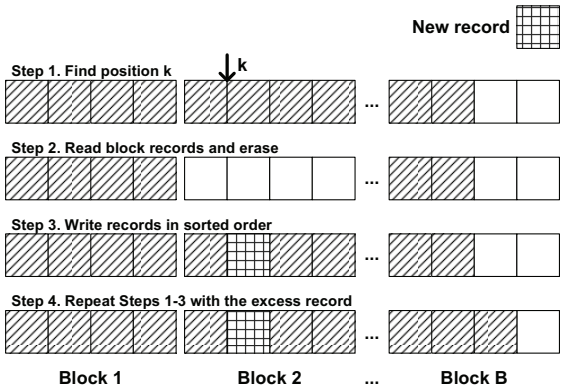


Figure 1: Example of Online Sorting

- **Wear-Constraint:** The number of times a block can be written or erased is limited.
- **Fast Read/Slow Write-Constraint:** Whereas typically read and write access on a hard disk are almost identical, in flash it is much faster to read ($\approx$60μs) rather than write ($\approx$800μs).

Now that we have seen the characteristics/constraints of flash memory, we will respond on which of the two sorting approaches (online vs. offline) is superior by performing some preliminary analysis. Let us consider the example depicted in Figure 1 which illustrates a flash memory consisting of B blocks each containing P=4 pages. At some time τ the flash memory contains N elements distributed throughout the blocks in sorted order. Let us assume that on time $\tau+1$ a new element arrives and needs to be stored in sorted order. The first step is to discover the position k that the new element must be inserted (this requires log_N reads). When k is found, the block containing k is first read into memory (P reads) and then erased (due to the Erase-Constraint) in order to update it. If the block contains $\geq$1 empty pages then the procedure halts, otherwise the procedure continues to the next block with the excess page at hand. This technique is extremely inefficient to flash memory as it consists of too many write/erase operations that decrease both the performance and wearability of flash memory. In fact, we will demonstrate this level of detriment using a series of microbenchmarks in Section 5.

On the other hand, offline sorting requires the input be sorted only when requested, thus using less write/erase operations. This is generally preferable in cases where writing to memory is significantly more expensive than reading like flash memory. Our FSort algorithm extends the basic principles of offline sorting to further enhance its performance by minimizing the write/erase operations. This provides efficient access to the data stored on flash memory while in parallel increases the longevity of the flash memory by spreading page writes out uniformly so that the available storage capacity does not diminish at particular regions of the flash media.

Our Contributions

In this paper we make the following contributions:

- We devise a sorting algorithm, coined FSort that intelligently exploits the characteristics of flash memory to deliver good performance both in terms of time and energy.;
- We provide an extensive experimental evaluation using traces from a real sensor network deployment at Intel Research Berkeley [6].

NAND-based flash memory installed on a WSD			
	Page **Read** 1.17mA	Page **Write** 37mA	Block **Erase** 57mA
Time	6.25ms	6.25ms	2.26ms
Energy	24μJ	763μJ	425μJ
	Page **Erase-Write** 43mA	Flash **Idle** 0.068mA	Flash **Sleep** 0.031mA
Time	6.75ms	N/A	N/A
Energy	957μJ	220μJ/sec	100μJ/sec

Table 1: Performance Parameters for NAND-based flash memory using a 3.3V voltage, 512B Page size and 16KB Block size

The remainder of the paper is organized as follows: Section 2 overviews the related research work and Section 3 formalizes our system model and assumptions. Section 4 introduces the FSort algorithm with its two phases. Next, in Section 5 we present our experimental study and finally Section 6 concludes our paper.

2. BACKGROUND AND RELATED WORK

2.1 Flash Memory

Flash is a new generation of non-volatile memory that introduces many advantages compared to traditional storage media, including: shock-resistance, fast read access, low production cost and power efficiency. Flash memory is nowadays the de-facto storage medium for WSDs and a variety of other mobile devices.

There exists two types of flash memory, namely *NOR* and *NAND* (both names where chosen because the internal structure resembled the respective gates) NOR flash characteristics are fast read access, slow write/erase time, low density and a random-access interface. The latter property makes NOR flash ideal for application or boot code execution as it allows direct access any address location. On the other hand NAND-flash has faster write/erase time and requires a smaller chip area per cell; thus increasing the overall storage capacity; this also lowers the cost of NAND flash. However, NAND flash does not allow random access to any memory location like NOR flash.

2.2 Flash-based Databases

In this section we briefly describe some of the most popular approaches to flash-based databases that try to cope with the constraints mentioned in Section 1.

- **Flash Translation Layer (FTL) approach:** One way to deploy traditional hard disk based database software on flash memory is by utilizing a *Flash Translation Layer* (FTL) [11]. FTL maintains an internal mapping table between logical and physical pages that translates write request to flash memory. This approach simplifies the deployment of traditional database software but it also introduces a significant overhead

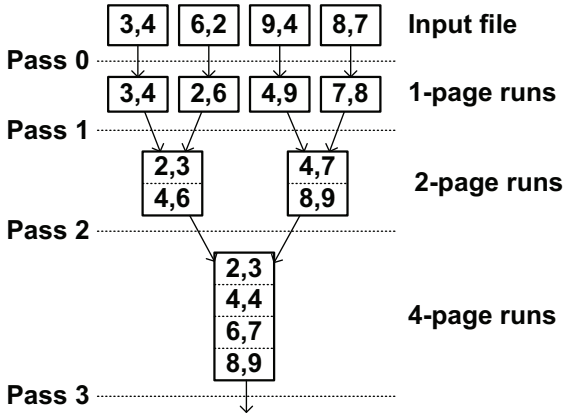


Figure 2: External Sorting example with M=3 (2 input and 1 output page buffers).

on flash memory as most popular database systems require frequent small log writes to store the old and updated values of data.

- **Log Structured approach:** The *Log Structured approach* (LSA) [18] differs from FTL as it enables databases to access flash memory directly. LSA considers the database as one large log and any update to the database (i.e., log) is appended at the end of flash memory. This approach uniformly distributes write requests on flash thus improving both performance and wearability. However, like FTL, LSA suffers from the existence of frequent small log writes. Additionally, since only log pages are written on flash, when a database is requested it has to be recreated by reading all associated log pages.

- **In-page Logging approach:** A more recent approach, coined *In-page Logging* (IPL), that tries to overcome the problems of LSA was introduced in [10]. IPL stores the log records associated with a specific database page on the same block thus allowing faster recreation of the database page. However, frequent small writes also affect the IPL approach.

- **AceDB flashlight:** AceDB [9] tries to overcome the small log frequent writes problem by maintaining a logical to physical page map in RAM. Additionally, physical pages are allocated to logical pages on a transaction basis which allows for more sequential writes within a block.

In this work, we assume that the data is structures on flash media using a modified version of the IPL approach. In this approach, new records arising from continuous queries are maintained in a buffer and when the buffer reaches the size of a page it is appended sequentially at the end of the flash, i.e., the next available empty page. This scheme is highly beneficial as it eliminates frequent small writes to flash media. Additionally this scheme does not fragment flash memory as it operates in a top-down, left-to-right manner which satisfies the Write-Constraint B.

2.3 External Sorting

Symbol	Definition
N	Flash capacity
M	Available main memory (pages)
B	Number of Blocks in flash
b_i	Block with identifier $i : 1 \leq i \leq B$
P	Number of Pages in a Block
p_i^j	Page with identifier i in block j

Table 2: Definition of Symbols

Sorting is one of the fundamental problems of computer science and has been extensively studied by the research literature. External sorting arises when the number of records to be sorted is larger than the available main memory of the system. The fundamentals of external sorting, including replacement selection and polyphase merging strategies have been thoroughly analyzed in [7] using tapes as the storage medium. Since then, many researches devised solutions to integrate and improve the performance of external sorting algorithms (I/O operations, memory requirements) on hard drives disks [17, 16, 8].

Traditional external sorting algorithms usually consist of two phases i) *internal sorting*, and ii) *external merging*. In the internal sorting phase, the input file (size S pages) is streamed into main memory, sorted on a page basis and written on to flash (Pass 0). The resulting sorted pages are called *runs* [7]. As soon as the runs are produced, the external merging phase starts. Assuming an available memory of size M (pages), the external merging phase utilizes 1 page for output and M-1 pages for input buffers. On each subsequent pass, M-1 runs are fed into the input buffers and are merged using the output buffer to produce longer runs. The output buffer is written out to flush and emptied so as to be used in the next pass. Figure 2 illustrates an example of external sorting with M=3 (2 input and 1 output page buffers).

3. SYSTEM MODEL

In this section we will formalize our basic terminology and assumptions. The main symbols and their respective definitions are summarized in Table 2. For brevity in our descriptions let us denote *NAND-based flash memory* as *flash*. Flash contains B blocks $(b_1, b_2, ..., b_B)$ each containing P pages. We define p_i^j as page with identifier $i : 1 \leq i \leq B$ that belongs to block b_j. We also assume that the WSD has a maim memory module of capacity M (M pages). According to the modified IPL approach, each time a WSD acquires measurements from the sensor-board it stores them sequentially on flash in a top-down, left-to-right manner. In our experiments, we assume that a measurement is acquired every 1ms.

At periodic time intervals (e.g., every 1024ms) $\tau_1, \tau_2, ...$, a user posts a query Q to the sensor network that requires sorted data. Upon reception of Q a sensor sorts the locally stored data and transmits them. Note that the amount of data, X, sorted each time is linearly correlated with τ_i, i.e., $\tau_1 = X, \tau_2 = 2X, \tau_3 = 3X, ...$ and so on.

As described earlier in Section 1, our FSort algorithm consists of two phases, namely internal sorting and external merging. During the internal sorting phase M-1 buffers are created in memory that access blocks in a top-down, left-to-right manner. These buffers are used to create the initial

runs which are always appended at the end of flash. As soon as the first phase is completed the external merging phase produces the sorted output by merging the runs recursively.

4. THE FSORT ALGORITHM

In this section we describe our FSort algorithm. FSort consists of two phases:

1. **Internal Sorting phase:** This phase produces sorted runs by streaming data pages into memory, sorting them using a top-down replacement selection algorithm and outputting and appending them at the end of flash memory.

2. **External Merging phase:** During this phase, the sorted runs that were generated in the internal sorting phase, are recursively merged until a single sorted output is produced.

Our initial sorting phase algorithm accesses flash memory in a top-down manner by initiating a page buffer array in available memory. Since in WSDs the amount of available memory is very limited we have chosen to set the size of the buffer array equal to the number of pages in a block (usually 8-16) [5, 4]. This enables us to use linear control flow structures that speed up operations. Data pages are continuously traversed until all input is examined.

The intuition behind our approach is that the measurements generated by continuous queries in WSNs are not altered erratically in small time windows. As a result, our approach generates longer runs on each pass thus minimizing the overall number of passes; decreasing in this way both read, write and erase operations. Furthermore, accessing pages in a top-down manner frees up empty space at the top (as soon as all pages in a block have been fed to the buffer array the block can be erased). This is particularly useful as erasing individual pages is not permitted in flash memory due to the Erase-Constraint mentioned in Section 1. Algorithm 1 outlines the operation of the internal sorting phase of FSort.

Assuming that we have D unsorted data pages and an available main memory M, our algorithm starts out by initiating a P-page buffer array, coined Buf, that will be used to traverse the flash memory in a top-down manner (line 2). We also set the counter $CountEmpty$ to zero (line 3);this counter will be used to halt the procedure when all buffers are accessing empty ore new pages (i.e., we have read all data pages). The algorithm continues with streaming the pages of the first block into the Buf array (lines 5-7). Next, the algorithm proceeds continuously by selecting each time the smallest key (or highest depending on the request) including the buffer index, $BIndex$, that the key was discovered (line 10) and forwards it to the output buffer (line 11). This is done efficiently by utilizing a selection tree. As soon as the key is outputted onto flash we need to retrieve the next page for Buf_{Bindex}. To do this, we store in the header of the buffer the current page position and assign it to a temporary variable $PIndex$ while in parallel increasing it by 1 (to access the next page) (line 13-14). If the page pointed by $PIndex$ is empty or if a sorted page has been outputted to this position (line 16) then we simply increase counter $CountEmpty$ (line 19); this also removes the Buffer from the selection tree, otherwise we load the new page to the current buffer (line 17). Finally, we test if there exists more than one buffer accessing data pages and if not we exit the loop (line 21-24). The procedure ends by recursively merging the runs (line 26).

Algorithm 1 : **FSort**

Input: D unsorted data pages, Available Flash Memory M, Buf:$P = (M-1)$ input buffers, min:1 output buffer
Output: D **sorted** data pages

```
1: procedure SORT
2:     Allocate(Buf(P)); //Allocate P buffers for input
3:     CountEmpty=0; //Terminating counter
4:     //Initiate the Buf buffer array
5:     for i = 1 to P do
6:         set Buf_i = p_{i%P}^{i/P};
7:     end for
8:     while true do
9:         //find the smallest key (min) and its index in Buf
10:        set (min, BIndex) = SelectionTree(Buf);
11:        Output(min); //write sorted output to flash
12:        //find the next page that the Buf_BInx must retrieve
13:        set Buf_BIndex->lastPageRead+ = P;
14:        set PIndex = Buf_BIndex->lastPageRead;
15:        //check if empty or new page
16:        if (!IsEmptyOrNew(p_{PIndex}^{BIndex}) then
17:            set Buf_BIndex = p_{PIndex}^{BIndex};
18:        else
19:            set CountEmpty++;
20:        end if
21:        if (CountEmpty == P) then
22:            //all buffers read empty pages
23:            Break;
24:        end if
25:    end while
26:    Merge();
27: end procedure
```

As soon as the internal sorting phase begins, the sorted runs are merged into longer runs by the external merging phase. This procedure continuous recursively and terminates when a single sorted run is generated.

5. EXPERIMENTS

In this section we present an extensive experimental comparison of the FSort algorithm.

5.1 Experimental Methodology

Power Model: We adopt the power model of the RISE platform [25] which consists of the following parameters: We use a 14.8 MHz 8051 core operating at 3.3V with the following current consumption 14.8mA (On), 8.2mA (Idle), 0.2ţA (Off). We utilize a 128MB flash media with a page size of 512B and a block size of 16KB. The current to read, write and block delete was 1.17mA, 37mA and 57μA and the time to read in the three pre-mentioned states was 6.25ms, 6.25ms, 2.27ms.

Datasets: We utilize a real trace of sensor readings that is collected from 58 sensors deployed at the premises of the Intel Research in Berkeley [6] between February 28th and April 5th, 2004. The sensors utilized in the deployment were equipped with weather boards and collected time-stamped topology information along with humidity, temperature, light and voltage values once every 31 seconds (i.e., the epoch). The dataset includes 2.3 million readings collected from these sensors.

Power (mJ)		Read	Write	Erase
NAND-flash	Online	47815	1519376	105789
	Offline	96	3052	213
NOR-flash	Online	5998	189921	48279
	Offline	96	3052	776
Time (s)		Read	Write	Erase
NAND-flash	Online	12452	12446	563
	Offline	25	25	1
NOR-flash	Online	1562	1546	122
	Offline	25	25	2

Table 3: Online vs. Offline Sorting Performance

Algorithms: We implement the FSort algorithm and an optimized version of the traditional external mergesort algorithm that utilizes M-1 buffer pages for input and one for output.

5.2 Experimental Results

In this Section we present the results of our experimental study.

A. Online vs. Offline Sorting on flash

In our first experimental series we have conducted a micro benchmark that shows the deficiencies of online sorting on flash memories used in WSDs. We have implemented: i) the InsertionSort algorithm, and ii) the External MergeSort algorithm for benchmarking the online and offline sorting approaches respectively. We feed 1000 records (1 record per page) to our simulator and record the number of read, write and erase operations which we then translate to power and time consumption.

The results for both NOR and NAND-based flash memories are depicted on Table 3. We observe that the offline sorting approach greatly outperforms the online approach on NAND and NOR-based flash memory both with regards to power and time. Specifically, we observe a decrease of power consumption and time by 3 orders of magnitude on NAND-based flash memory and 2 orders of magnitude on NOR-based flash memory.

B. Power Consumption

In our second experimental series we measure the energy performance of FSort and compare it with the external Mergesort with M-1 input buffers. We run experiments using three different attributes included in our dataset, namely temperature, humidity and light. The sensor records one attribute every 1ms and a query requests sorted results every 1024ms. Our simulator executes both the external MergeSort and FSort algorithms every 1024ms and collects energy statistics every 1024=1K records.

Figure 3 illustrates the results of our evaluation with regards to energy consumption[1] for the light attribute; similar results apply to the humidity and temperature attributes. We observe that FSort always maintains a competitive advantage over MergeSort for all data sizes (1-32K). The initial energy savings for 1K are ≈25% but as the number of records increases the energy savings also increase reaching as high as ≈41%. This is augmented to the fact that FSort minimizes write and erase operations by releasing continuous blocks of pages every time which are the most expensive operations; this also increases the longevity of flash memory.

[1]We omit the results of time performance as they are highly correlated with energy consumption and do not present any new findings.

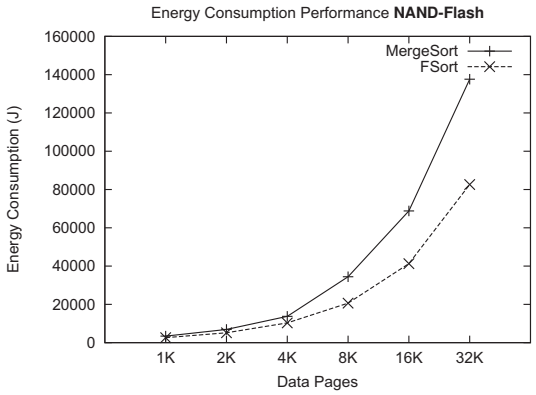


Figure 3: Energy Consumption performance comparison of MergeSort and FSort for the light attribute. Sort occurs every 1K records.

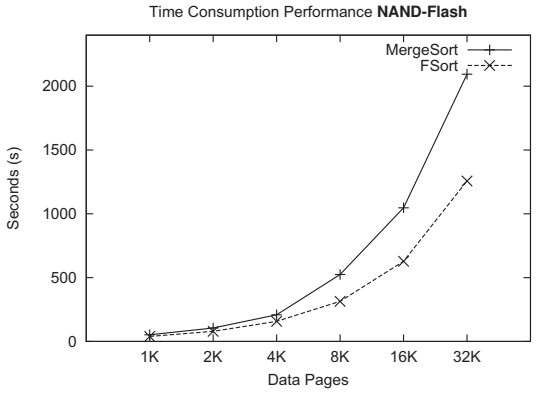


Figure 4: Time performance comparison of MergeSort and FSort for the light attribute. Sort occurs every 1K records.

C. Time Consumption

In our last experimental series we assess the efficiency of our FSort algorithm with regards to time performace. Like the previous setting sensor records one attribute every 1ms and a query requests sorted results every 1024ms. Our simulator executes both the external MergeSort and FSort algorithms every 1024ms and collects time statistics every 1024=1K records.

Figure 4 illustrates the results of our evaluation with regards to energy consumption for the light attribute; similar results apply to the humidity and temperature attributes. We observe that FSort always maintains a competitive advantage over MergeSort for all data sizes (1-32K). The initial energy savings for 1K are ≈25% but as the number of records increases the energy savings also increase reaching as high as ≈41%. This is augmented to the fact that FSort minimizes write and erase operations by releasing continuous blocks of pages every time which are the most expensive operations;

this also increases the longevity of flash memory.

6. CONCLUSIONS

In this paper we proposed a novel flash-aware and power-efficient algorithm coined FSort. FSort takes into account the limitations of flash memory used in WSDs and improves both overall execution and response time performance.

Our experiments with real traces show that FSort, greatly outperforms the traditional external mergesort approach by minimizing write and erase operations which are the most expensive operations on flash. In the future we plan to extend our work by including indexes to access the data more efficiently as well as perform experiments on other types of flash memory like SSDs.

7. REFERENCES

[1] Aly M., Chrysanthis P.k., Pruhs K., "Decomposing Data-Centric Storage Query Hot-spots in Sensor Networks", In *MOBIQUITOUS*, 2006.

[2] Aly M., Pruhs K., Chrysanthis P.k., "KDDCS: a load-balanced in-network data-centric storage scheme for sensor networks", In *CIKM* pp.317-326, 2006.

[3] Andreou P., Zeinalipour-Yazti D., Vassiliadou M., Chrysanthis P.K., Samaras G., "KSpot: Effectively Monitoring the K Most Important Events in a Wireless Sensor Network", In *ICDE*, 2009.

[4] Atmel AT45DB041B, http://www.atmel.com/

[5] Crossbow Technology Inc., http://www.xbow.com/

[6] Intel Lab Data http://db.csail.mit.edu/labdata/labdata.html

[7] Knuth D.E., "The Art of Computer Programming: Sorting and Searching" Addison Wesley, Vol. 3, April 1997, ISBN:0-201-89685-0.

[8] Larson P., Graefe G., "Memory management during run generation in external sorting", In *ACM SIGMOD*, pp.472-483, 1998.

[9] Lee K.Y., Kim H., Woo K., Chung Y.D., Kim M.H., "Design and implementation of MLC NAND flash-based DBMS for mobile devices", In *Journal of Systems and Software*, 2009.

[10] Lee S., Moon B., "Design of Flash-Based DBMS: An In-Page Logging Approach", In *ACM SIGMOD*, pp.1-10, 2007.

[11] Lee S., Park D., Chung T., Lee D., Park S., Song, H., "A log buffer-based flash translation layer using fully-associative sector translation", In *ACM (TECS)*, Vol.6, No.3, pp.18, 2007.

[12] Madden S.R., Franklin M.J., Hellerstein J.M., Hong W., "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", In *USENIX OSDI*, Vol.36, pp.131-146, 2002.

[13] Masuoka F., Iizuka H., "Semiconductor memory device and method for manufacturing the same ", US patent:4531203, July,1985.

[14] Elmasri R., Navathe S.B., "Fundamentals of Database Systems (5^th Edition):Chapter:15.3.2" Addison Wesley, 2007, ISBN:0-321-41506-X.

[15] Ning X., Sumit R., Krishna K.C., Deepak G., Alan B., Ramesh G., Deborah E., "A wireless sensor network For structural monitoring", In *ACM SenSys*, pp.13-24, 2004.

[16] Nyberg C., Barclay T., Cvetanovic Z., Gray J., Lomet D., "AlphaSort: a cache-sensitive parallel external sort", In *VLDB*, Vol.4, No.4, pp.603-628, 1995.

[17] Pang H., Carey M.J., Livny M., "Memory-Adaptive External Sorting", In *VLDB*, pp.618-629, 1993.

[18] Rosenblum M., Ousterhout J.K., "The design and implementation of a log-structured file system", In *ACM TOCS*, Vol.10, No.1, pp.26-52, 1992.

[19] Sadler C., Zhang P., Martonosi M., Lyon S., "Hardware Design Experiences in ZebraNet", In *ACM SenSys*, pp.227-238, 2004.

[20] Selavo L., Wood A., Cao Q., Sookoor T., Liu H., Srinivasan A., Wu Y., Kang W., Stankovic J., Yound D., Porter J., "LUSTER: wireless sensor network for environmental research", In *ACM SenSys*, pp.103-116, 2007.

[21] Sharaf M.A. , Beaver J., Labrinidis A. and Chrysanthis P.K., "TiNA: a scheme for temporal coherency-aware in-network aggregation", In *MobiDe*, pp.69-76, 2003.

[22] Szewczyk R., Mainwaring A., Polastre J., Anderson J., Culler D., "An Analysis of a Large Scale Habitat Monitoring Application", In *ACM SenSys*, pp.214-226, 2004.

[23] Yao Y., Gehrke J.E., "The cougar approach to in-network query processing in sensor networks", In *SIGMOD Record*, Vol.32, No.3, pp.9-18, 2002.

[24] Zeinalipour-Yazti D., Andreou P., Chrysanthis P. and Samaras G., "MINT Views: Materialized In-Network Top-k Views in Sensor Networks", In IEEE *MDM*, 2007.

[25] Zeinalipour-Yazti D., Lin S., Kalogeraki V., Gunopulos D., Najjar W., "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices", In *Usenix FAST*, pp.31-44, 2005.