ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Ένας Πυρήνας ASIC για Διαχείριση Ουρών Προτεραιότητας με τη Χρήση της Τεχνικής της Ομοχειρίας (Pipelining), για Υποστήριξη Χρονοδρομολόγησης σε Δίκτυα Υψηλών Ταχυτήτων

Άγγελος Δ. Ιωάννου

Μεταπτυχιακή Εργασία

Ηράκλειο Κρήτης, Οκτώβριος 2000

Ένας Πυρήνας ASIC για Διαχείριση Ουρών Προτεραιότητας με τη Χρήση της Τεχνικής της Ομοχειρίας (Pipelining), για Υποστήριξη Χρονοδρομολόγησης σε Δίκτυα Υψηλών Ταχυτήτων

Μεταπτυχιακή Εργασία
που υποβλήθηκε στην επιτροπή μεταπτυχιακών σπουδών
του Τμήματος Επιστήμης Υπολογιστών
της Σχολής Θετικών Επιστημών
του Πανεπιστημίου Κρήτης
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση του
ΔΙΠΛΩΜΑΤΟΣ ΜΕΤΑΠΤΥΧΙΑΚΗΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Άγγελος Δ. Ιωάννου,
Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Μανόλης Κατεβαίνης, Επόπτης
(Καθηγητής)

Ευάγγελος Μαρκάτος, Μέλος
(Επίκουρος Καθηγητής)

Απόστολος Τραγανίτης, Μέλος
(Αναπληρωτής Καθηγητής)

Δεκτή:

Πάνος Κωνσταντόπουλος,
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών
(Καθηγητής)

Ηράκλειο Κρήτης, Οκτώβριος 2000

Ένας Πυρήνας ASIC για Διαχείριση Ουρών Προτεραιότητας με τη Χρήση της Τεχνικής της Ομοχειρίας (Pipelining), για Υποστήριξη Χρονοδρομολόγησης σε Δίκτυα Υψηλών Ταχυτήτων

Άγγελος Δ. Ιωάννου
Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

ΠΕΡΙΛΗΨΗ

Εγγυήσεις για ποιότητα εξυπηρέτησης (QoS) σε δίκτυα θα προσφέρονται σύντομα με τη χρήση ουρών και προηγμένης χρονοδρομολόγησης (scheduling). Οι περισσότεροι προηγμένοι αλγόριθμοι χρονοδρομολόγησης στηρίζονται σε ένα κοινό υπολογιστικό μέρος: τις ουρές προτεραιότητας. Μεγάλες ουρές προτεραιότητας φτιάχνονται με δομές δεδομένων όπως οι σωροί (heaps). Για να υποστηρίξουμε προηγμένη χρονοδρομολόγηση σε ρυθμούς OC-192 (10 Gbps) και πάνω, χρήση της τεχνικής της ομοχειρίας (pipelining) απαιτείται για τη διαχείριση της ουράς προτεραιότητας. Παρουσιάζουμε ένα διαχειριστή σωρού (heap) που κάνει χρήση της τεχνικής ομοχειρίας, τον οποίο έχομε σχεδιάσει σαν ένα πυρήνα (core) τοποθετίσιμο (integradable) κυκλώματα ASIC, και τον οποίο έχομε περιγράψει σε μορφή συνθέσιμης (synthesizable) Verilog. Συζητούμε πώς μπορεί να χρησιμοποιηθεί σε switches και routers, τα πλεονεκτήματά του σε σχέση με τις ουρές ημερολογίου (Calendar Queues), και αναλύουμε εναλλακτικές λύσεις για κόστος-απόδοση. Χρησιμοποιώντας δίπορτες και τετραπλού πλάτους μνήμες (SRAM) και οικουμενικά προσπεράσματα (global bypasses), οι εντολές-λειτουργίες μπορούν να ξεκινούν με ρυθμό μίας ανά κύκλο ρολογιού. Όταν μειώνεται το κόστος, ο ρυθμός αποδοχής εντολών ελλατώνεται σε μία ανά κάποιο μικρό αριθμό κύκλων. Το σύστημα μπορεί να τροποποιείται για οποιοδήποτε μέγεθος σωρού και υποστηρίζει έναρξη μιας εντολής ανά κύκλο ρολογιού, εκτός αν έχομε συνεχόμενες εντολές διαγραφής (delete), όπου χρειάζεται ένας κενός (idle) κύκλος ρολογιού για να τις διαχωρίζει. Έχομε επαληθεύσει (verified) το σύστημά μας, συν-προσομοιώνοντάς το με τρία μοντέλα σωρών διαφορετικής πολυπλοκότητας και αφαίρεσης. Έχομε επίσης εκτελέσει synthesis και παρουσιάζομε πληροφορίες για την ανάλυση κόστους.

An ASIC Core for Pipelined Heap Management to Support Scheduling in High Speed Networks

Aggelos D. Ioannou
Master of Science Thesis

Department of Computer Science
University of Crete

ABSTRACT

Quality-of-Service (QoS) guarantees in networks will soon be provided using per-flow queueing and sophisticated schedulers. Most advanced scheduling algorithms rely on a common computational primitive: priority queues. Large priority queues are built using calendar queue or heap data structures. To support advanced scheduling at OC-192 (10 Gbps) rates and above, pipelined management of the priority queue is needed. We present a pipelined heap manager that we have designed as a core integratable into ASIC's, in synthesizable Verilog form. We discuss how to use it in switches and routers, its advantages over calendar queues, and we analyze the cost-performance tradeoffs: using 2-port, 4-wide SRAM's and global bypasses, heap operations can be initiated at the rate of one per clock cycle; when reducing these costs, issue rates drop to one every few cycles. Our design can be configured to any heap size, and supports initiating operations on every clock cycle, except that one idle (bubble) cycle is needed between two successive delete operations. We have verified our design by co-simulating it with three heap models of varying abstraction. We have also performed synthesis, and are presenting cost analysis information.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω όλους αυτούς που με βοήθησαν κατά τη διάρκεια της εργασίας μου. Πρώτα από όλους θα ήθελα να ευχαριστήσω τον καθηγητή κ. Μανόλη Κατεβαίνη που έδωσε την ιδέα για την όλη εργασία. Ακόμη θα ήθελα να τον ευχαριστήσω τόσο για τις ανεκτίμητες προτάσεις και ιδέες κατά τη διάρκεια της εργασίας, όσο και για την προσπάθειά του να μου αποκαλύψει και να μου μεταφέρει τον τρόπο με τον οποίο πρέπει κάποιος να σκεφτεται όταν σχεδιάζει, και γενικότερα όταν απασχολείται με επιστημονική έρευνα.

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον Γεώργιο Κορνάρο που με βοήθησε με τα εργαλεία σχεδιασμού hardware της CADENCE, τόσο με την ενασχόλησή του για τη σύνθεση του heap-manager, όσο και για την εκμάθηση των εργαλείων αυτών σε εμένα.

Επίσης τον καθηγητή Διονύσιο Πνευματικάτο για τις βοηθητικές του παρατηρήσεις για την συγγραφή synthesizable – και μάλιστα αποδοτικά synthesizable – κώδικα verilog, και τους καθηγητές Ε. Μαρκάτο και Α. Τραγανίτη που συμμετείχαν για την αξιολόγηση αυτής της εργασίας.

Θα ήθελα να ευχαριστήσω τη Europractice και το Πανεπιστήμιο Κρήτης για την παροχή πολλών από τα εργαλεία CAD που χρησιμοποιήθηκαν, και το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας καθώς και την Γενική Γραμματεία Έρευνας και Τεχνολογίας Ελλάδος για την οικονομική υποστήριξη αυτής της εργασίας.

Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου για την βοήθεια και την υποστήριξη κατά τη διάρκεια των σπουδών μου.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

1 Εισαγωγή

Η ταχύτητα των δικτύων αυξάνεται με δραματικούς ρυθμούς. Συμαντικές πρόοδοι γίνονται ακόμη στην αρχιτεκτονική των δικτύων, και ειδικά στην παροχή εγγύησεων ποιότητας υπηρεσιών (Quality of Service). Μεταγωγείς (Switches) και δρομολογητές (routers) βασίζονται όλο και πιο πολύ σε ειδικό υλικό (hardware) για την παροχή του απαιτούμενου υψηλού throughput και προηγμένου QoS. Τέτοιο hardware υποστήριξης γίνεται εφικτό και οικονομικό με τις προόδους στην τεχνολογία ημιαγωγών. Για να μπορούν να παρέχουν υψηλού επιπέδου εγγυήσεις QoS, τα switches και οι routers ενός δικτύου χρειάζονται *ουρές ανά ροή (per-flow queueing)* και *προηγμένη χρονοδρομολόγηση (advanced scheduling)* [Kumar98]. Το θέμα αυτής της εργασίας είναι η υποστήριξη με hardware για προηγμένη χρονοδρομολόγηση.

Το per-flow queueing αναφέρεται στην αρχιτεκτονική όπου τα πακέτα που ανταγωνίζονται και περιμένουν την αποστολή σε ένα συγκεκριμένο link εξόδου, τοποθετούνται σε πολλαπλές ουρές. Στα εναλλακτικά –single-queue systems– ο τρόπος εξυπηρέτησης είναι αναγκαστικά first-come-first-served (FCFS), ο οποίος στερείται δικαιότητας και απομόνωσης μεταξύ των σωστά και των λάθος συμπεριφερόμενων flows. Μια μερική λύση είναι η χρήση FCFS, αλλά να βασιστούμε σε έλεγχο της κυκλοφορίας στις πηγές, βασισμένο σε συμβόλαια εξυπηρέτησης (admission control) ή σε προτόκολα για end-to-end flow control (όπως το TCP). Ενώ αυτά μπορούν να επιτύχουν δίκαιη κατανομή του throughput σε μεγάλο χρονικό διάστημα, υποφέρουν από μη-αποδοτικότητα σε σύντομα χρονικά διαστήματα: όταν ένα καινούργιο flow ζητάει μερίδιο του throughput, τότε υπάρχει καθυστέρηση στο να επιβραδυνθούν τα παλιά flows, και αντίστροφα, όταν ένα νέο flow ζητάει τη χρήση μη χρησιμοποιούμενων (idle) στοιχείων, υπάρχει καθυστέρηση εώς ότου κάτι τέτοιο επιτραπεί. Σε μοντέρνα δίκτυα υψηλού throughput, αυτές οι κυκλικές (round-trip) καθυστερήσεις –υπαρκτές σε κάθε σύστημα ελέγχου– αντιστοιχούν σε συνεχώς αυξανόμενο αριθμό πακέτων.

Για την παροχή πραγματικής απομόνωσης μεταξύ των flows, τα πακέτα του καθένα θα πρέπει να περιμένουν σε ξεχωριστές ουρές, και ένας scheduler πρέπει να εξυπηρετήσει τις ουρές με μια σειρά που να δίνει δίκαια το διαθέσιμο throughput στα ενεργά flows. Σημειώστε πως δικαιότητα δεν σημαίνει κατ'ανάγκη ισότητα –διαφοροποίηση της εξυπηρέτησης, με ελεγχόμενο τρόπο, είναι μια απαίτηση για τον scheduler. Εμπορικά switches και routers ήδη χρησιμοποιούν πολλαπλές ουρές ανά έξοδο, αλλά ο αριθμός των ουρών αυτών είναι μικρός (μερικές δεκάδες ή μικρότερος), και ο scheduler που τις διαχειρίζεται είναι σχετικά απλός (π.χ. απλές προτεραιότητες). Το μέλλον θα φέρει πιο λεπτομερή διαχωρισμό, και τότε πολλές χιλιάδες ουρών θα απαιτούνται. Η διατήρηση πολλών ουρών και η διαχείρισή τους σε υψηλή ταχύτητα προϋποθέτει ένα όχι αμελητέο κόστος, αλλά είναι εφικτή με τη μοντέρνα τεχνολογία ημιαγωγών [Korn97].

Η παρούσα εργασία ασχολείται με το εξής πρόβλημα: την υλοποίηση προηγμένων αλγόριθμων scheduling σε υψηλή ταχύτητα, όταν υπάρχουν πολλές χιλιάδες από ανταγωνιζόμενα flows. Η ενότητα 2 παρουσιάζει μια σύνοψη των διάφορων αλγόριθμων scheduling. Όλοι βασίζονται σε ένα κοινό υπολογιστικό μέρος για την πιο χρονοβόρα διαδικασία τους: η εύρεση του μέγιστου (ή του ελάχιστου) ανάμεσα από ένα μεγάλο πλήθος τιμών. Προηγούμενη δουλειά στην υλοποίηση αυτού του μέρους δίνεται στο [Ioan00, section 2]. Για μεγάλο αριθμό ευρέως διεσπαρμένων τιμών, οι ουρές προτεραιότητας σε μορφή της *δομής δεδομένων του σωρού (heap data structure)* είναι η πιο αποδοτική αναπαράσταση, παρέχοντας εντολές εισαγωγής (insert) και διαγραφής ελαχίστου (delete) σε λογαριθμικό χρόνο. Για heap αρκετών εκατοντάδων στοιχείων, αυτό μεταφράζεται σε μερικές δεκάδες προσπελάσεις σε μια RAM για κάθε εντολή στο heap, το οποίο δίνει ένα ρυθμό εντολών μεταξύ 5 και 10 χιλιάδων εντολών το δευτερόλεπτο (Mops) ή και λιγότερο [Mavro98]. Όμως, για ρυθμούς OC-192 (10 Gbps) και παραπάνω, και για πακέτα σε μέγεθος περίπου 50 bytes, περισσότερα από 50 Mops απαιτούνται. Για να πετύχουμε τέτοιους ρυθμούς, οι εντολές του heap πρέπει να γίνονται pipelined. Τότε απαιτούνται μετατροπές στους απλούς (software) αλγόριθμους ενός heap, όπως είχε προταθεί το 1997 [Kate97] (δείτε επίσης το Technical Report [Mavro98]). Στην παρούσα εργασία ασχολούμαστε με έναν pipelined heap manager που σχεδιάσαμε στη μορφή ενός core, τοποθετίσιμο σε ASICs.

Η ενότητα 3 παρουσιάζει μια συνολική εικόνα του τρόπου λειτουργίας της δομής με την οποία ασχολούμαστε, αφού δόσει μια περιγραφή των δομικών της λίθων. Η περιγραφή αυτή έχει σκοπό να βοηθήσει στην συζήτηση που ακολουθεί. Στην ενότητα 4 παρουσιάζονται οι εντολές που θα μας απασχολήσουν, έτσι ώστε ο manager να παρέχει την αναγκαία λειτουργικότητα. Στην ενότητα 5 παρουσιάζονται οι αλγόριθμοι για γρήγορη είσοδο των εντολών.

Τελικά η ενότητα 6 παρουσιάζει την σχεδίασή μας, η οποία είναι σε μορφή synthesizable Verilog. Ο πυρήνας ASIC που σχεδιάσαμε μπορεί να μετατραπεί σε οποιοδήποτε μέγεθος ουράς προτεραιότητας. Τα datapaths παρουσιάζονται, ενώ κάποια προβλήματα για την υποποίηση των αλγορίθμων σε hardware και οι λύσεις τους παρουσιάζονται στα αντίστοιχα παραρτήματα (Appendices). Η απαλήθευση (verification) παρουσιάζεται στη συνέχεια, καθώς και το κόστος και η απόδοση του heap manager.

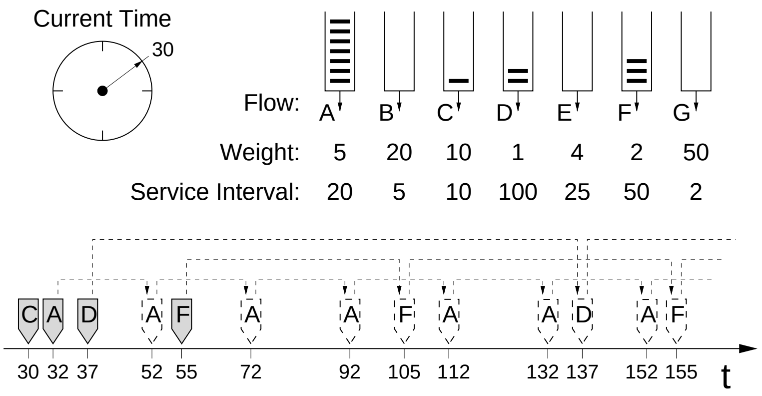
2 Προηγμένο Scheduling με τη Χρήση Ουρών Προτεραιότητας

Στην ενότητα 1 εξηγήσαμε την ανάγκη για per-flow queueing για την παροχή προηγμένου QoS σε δίκτυα υψηλών επιδόσεων του μέλλοντος. Για να είναι αποδοτικό, το per-flow queueing χρειάζεται ένα καλό scheduler. Πολλοί προηγμένοι αλγόριθμοι scheduling έχουν προταθεί. Καλές αναφορές αποτελούν τα [Zhang95] και [Keshav97, chapter 9]. Οι προτεραιότητες είναι ένας πρώτος, σημαντικός μηχανισμός. Συνήθως λίγα επίπεδα προτεραιότητας αρκούν, οπότε αυτός ο μηχανισμός υλοποιείται εύκολα. Το aggregation (ιεραρχικό scheduling) είναι ένας δεύτερος μηχανισμός: πρώτα διαλέγουμε μεταξύ ενός αριθμού από σύνολα flows, μετά διαλέγουμε ένα flow από το συγκεκριμένο σύνολο [Bennett97]. Μερικά επίπεδα της ιεραρχείας περιέχουν λίγα σύνολα, ενώ άλλα μπορεί να περιέχουν χιλιάδες flows. Η παρούσα εργασία αφορά τα τελευταία επίπεδα. Οι δυσκολότεροι αλγόριθμοι scheduling είναι αυτοί που ανήκουν στην οικογένεια weighted round robin. Ασχολούμαστε με αυτούς στη συνέχεια.

2.1 Η οικογένεια των Weighted Round Robin

Το *Σχ. 1* παρουσιάζει το weighted round robin scheduling. Βλέπουμε εφτά flows, από το A εως το G. Τέσσερα από αυτά, A, C, D, F είναι προς το παρόν ενεργά (μη-άδειες ουρές). Ο scheduler πρέπει να εξυπηρετήσει τα ενεργά flows με σειρά τέτοια ώστε η εξυπηρέτηση που θα δεχτεί το κάθε ενεργό flow σε μεγάλο χρονικό διάστημα να είναι ανάλογο του συντελεστή βάρους του (weight factor). *Δεν* είναι αποδεκτό να επισκεπτόμαστε τα flows με σειρά plain round robin, εξυπηρετώντας το καθένα αναλογικά με το βάρος του, γιατί οι χρόνοι εξυπηρέτησης για τα flows με μεγάλο βάρος θα συγκεντρωνόταν μαζί, οδηγώντας σε burstiness και σε μεγάλο jitter για το χρόνο εξυπηρέτησης.

Ξεκινούμε με τα ενεργά flows να είναι χρονοδρομολογημένα (scheduled) na εξυπηρετηθούν σε κάποιο συγκεκριμένο μελλοντικό 'χρόνο' το καθένα: Το flow A είναι να εξυπηρετηθεί στο χρόνο t=32, το C στο t=30, το D στο t=37, και το F στο t=55. Το flow που θα εξυπηρετηθεί επόμενο είναι αυτό που έχει το νωρίτερο, δηλ. το μικρότερο, scheduled χρόνο εξυπηρέτησης. Στο παράδειγμά μας αυτό είναι το flow C. Η ουρά του έχει μόνο ένα πακέτο, οπότε μετά την εξυπηρέτησή του γίνεται ανενεργό και διγράφεται από το schedule. Μετά, ο 'χρόνος' προχωράει στο 32 και εξυπηρετείται το flow A. Το flow A παραμένει ενεργό μετά τη μετάδοση του πρωτοπόρου πακέτου, και πρέπει να ξαναδρομολογηθεί (rescheduled). Το rescheduling βασίζεται στα αντίστροφα των συντελεστών



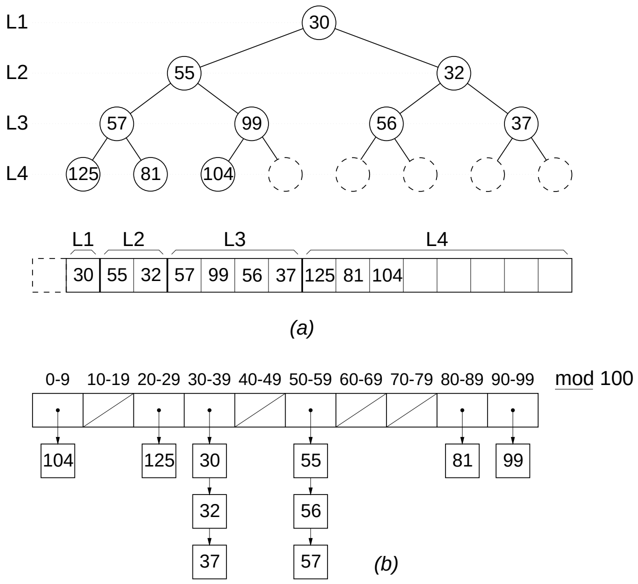
Σχήμα 1: Weighted round robin scheduling

βάρους, οι οποίοι αντιστοιχούν στα σχετικά διαστήματα εξυπηρέτησης. Το διάστημα εξυπηρέτησης του A είναι 20, οπότε το A γίνεται rescheduled για να εξυπηρετηθεί στο 'χρόνο' 32+20=52. (Όταν το μέγεθος πακέτου διαφέρει, αυτό το μέγεθος συμμετέχει επίσης στον υπολογισμό του επόμενου χρόνου εξυπηρέτησης). Η σειρά εξυπηρέτησης που προκύπτει φαίνεται στο *Σχ. 1* . Όπως βλέπουμε, ο scheduler λειτουργεί με το να ελέγχει την τιμή του "επόμενου χρόνου εξυπηρέτησης" για κάθε ενεργό flow. Σε κάθε βήμα πρέπει να βρούμε την ελάχιστη αυτών των τιμών, και μετά να την ενημερώσουμε (αυξήσουμε) αν το flow παραμάνει ενεργό, ή να τη διαγράψουμε αν το flow γίνεται ανενεργό. Όταν ένα νέο πακέτο ενός ανενεργού flow έρχεται, αυτό το flow θα πρέπει να επανεισαχθεί στο schedule.

Πολλοί αλγόριθμοι scheduling ανήκουν σε αυτή την οικογένεια. Όταν έχουμε work-conserving αρχές (disciplines) πάντα ο "τωρινός χρόνος" προωθείται στο χρόνο εξυπηρέτησης του επόμενου ενεργού flow, ανεξάρτητα με το πόσο μακριά στο μέλλον είναι αυτός. Με αυτό τον τρόπο, τα ενεργά flows απορροφούν όλη τη χωρητικότητα του δικτύου. Τα non-work-conserving disciplines χρησιμοποιούν ένα ρολόι πραγματικού χρόνου. Ένα flow εξυπηρετείται μόνο όταν το ρολόι πραγματικού χρόνου φτάσει ή ξεπεράσει τον scheduled χρόνο εξυπηρέτησης. Όταν δεν υπάρχει τέτοιο flow, ο μεταδότης μένει ανενεργός. Τέτοιοι schedulers λειτουργούν σαν 'ελεγκτές' (regulators), αναγκάζοντας το κάθε flow να υπακούει στο συμβόλαιο εξυπηρέτησής του. Άλλα σημαντικά στοιχεία ενός αλγόριθμου scheduling είναι ο τρόπος που ενημερώνει το χρόνο εξυπηρέτησης για το flow που εξυπηρετήθηκε (π.χ. βασισμένο στο διάστημα εξυπηρέτησης του flow, ή σε κάποιο deadline ανά πακέτο), και ο τρόπος που αποφασίζεται ο χρόνος εξυπηρέτησης για νεο-ενεργοποιημένα flows. Αυτά τα ζητήματα οδηγούν στις διαφορές μεταξύ των αλγορίθμων weighted fair queueing και των παραπλήσιων, όπως ο αλγόριθμος virtual clock και τα earliest-due-date και rate-controlled disciplines [Keshav97, ch.9].

2.2 Υλοποιήσεις Ουρών Προτεραιότητας

Όλοι οι παραπάνω αλγόριθμοι scheduling βασίζονται σε ένα κοινό υπολογιστικό μέρος για την πιο χρονοβόρα διεργασία τους: μια *ουρά προτεραιότητας*, δηλ. την εύρεση του μέγιστου (ή του ελάχιστου) από ένα σύνολο αριθμών. Ο ουρές προτεραιότητας με μόνο μερικές δεκάδες στοιχεία ή με αριθμούς προτεραιότητας επιλεγμένους από ένα μικρό σύνολο τιμών είναι εύκολο να υλοποιηθούν, π.χ. χρησιμοποιώντας συνδιαστικά κυκλώματα κωδικοποίησης προτεραιοτήτων (priority encoder circuits). Όμως, για ουρές προτεραιότητας με πολλές χιλιάδες στοιχεία και με τιμές επιλεγμένες από ένα μεγάλο σύνολο απιτρεπτών αριθμών, οι δομές δεδομένων τύπου *heap* ή *calendar queue* πρέπει να χρησιμοποιηθούν. Ά λλες δομές τύπου heap [Jones86] είναι ενδιαφέρουσες σε software αλλά δεν είναι προσαρμόσιμες σε υλοποιήσεις hardware υψηλής ταχύτητας.



Σχήμα 2: Priority queues: (a) heap; (b) calendar queue

Το *Σχ. 2* (α) παριστάνει ένα heap. Είναι ένα δυαδικό δέντρο (πάνω), φυσικά αποθηκευμένο σε ένα σειριακό πίνακα (κάτω). Τα μη-άδεια στοιχεία είναι όλα μαζεμένα πάνω και αριστερά. Το στοιχείο σε κάθε κόμβο είναι μικρότερο ή ίσο από τα στοιχεία των δύο θυγατρικών του κόμβων (η ιδιότητα του heap). Οι εισαγωγές (insertions) γίνονται στην αριστερότερη άδεια θέση, και μετά πιθανώς εναλλάσσονται με τους προγόνους των για να επανισχύσει η ιδιότητα του heap. Το ελάχιστο στοιχείο βρίσκεται πάντα στη ρίζα. Για να το διαγράψουμε (delete), μετακινούμε το τελευταίο γεμάτο στοιχείο στη ρίζα, και μετά πιθανώς το εναλλάσσουμε με τους απογόνους του που είναι μικρότεροί του. Στη χειρότερη των περιπτώσεων, θα χρειαστούν τόσες ανακατατάξεις όσο και το ύψος του δέντρου.

Το *Σχ. 2* (β) παριστάνει ένα calendar queue [Brown88]. Είναι ένας πίνακας με αλυσίδες. Τα στοιχεία τοποθετούνται στην αλυσίδα που υποδικνύεται από μια συνάρτηση διασποράς (hash function). Το επόμενο ελάχιστο στοιχείο βρίσκεται με το να ψάξουμε στην τωρινή αλυσίδα και μετά να ψάξουμε την επόμενη μη-άδεια αλυσίδα. Τα calendar queues έχουν καλή μέση απόδοση όταν το μέσο όρο υπολογίζεται σε μια μεγάλη σειρά εντολών. Όμως, βραχυπρόθεσμα, μερικές εντολές μπορεί να είναι πολύ ακριβές. Στο [Brown88], το calendar αλλάζει μέγεθος όταν η ουρά γίνεται πολύ μεγάλη ή πολύ μικρή. Αυτή η αλλαγή εμπεριέχει την αντιγραφή όλης της ουράς. Χωρίς αλλαγή του μεγέθους, είτε οι συνδεδεμένες λίστες θα γίνουν πολύ μακριές, καθυστερώντας τις εισαγωγές (αν οι λίστες είναι ταξινομημένες) ή το ψάξιμο (αν οι λίστες δεν είναι ταξινομημένες), ή οι σειρές άδειων στοιχείων στον πίνακα μπορεί να γίνουν πολύ μεγάλες, και θα χρειάζονται ειδική υποστήριξη για να ψάχνουμε το επόμενο μη-άδειο στοιχείο. Στο [Ioan00, section 2] υπάρχουν αναφορές σε προηγούμενη δουλειά πάνω σε υλοποιήσεις ουρών προτεραιότητας.

3 Η Δομή μας, Η Αναλοίωτη Συνθήκη της και Τα Στοιχεία της

Ας ξεκινήσουμε εξετάζοντας τι θα θέλαμε να προσφέρεται από μια αρχή χρονοδρομολόγησης (scheduling). Πρώτα απ'όλα, όπως αναφέρθηκε πριν, θα θέλαμε να έχουμε μια δομή που αποθηκεύει τα στοιχεία με τέτοιο τρόπο, ώστε πληροφορία σχετικά με έναν αριθμό διάταξης να μπορεί να εξαχθεί. Για να είμαστε πιο ακριβείς, θα θέλαμε να μπορούμε να πάρουμε το στοιχείο με τη μικρότερη προτεραιότητα στο μικρότερο δυνατό χρόνο, δηλαδή $O(1)$, επομένως ανεξάρτητο από τον αριθμό των στοιχείων (N) που είναι αποθηκευμένα στη δομή. Επομένως δεν απαιτείται συνολική διάταξη των στοιχείων. Όμως άλλες λειτουργίες για να κρατούν τη δομή ενημερωμένη θα πρέπει να γίνονται, όπως εισαγωγές και διαγραφές. Αυτή η διαχείτιση της δομής θα πρέπει επιπλέον να γίνεται αποδοτικά, έτσι ώστε να πετυχαίνουμε υψηλή συνολική απόδοση. Για να το πετύχουμε αυτό, μια μερική διάταξη των στοιχείων μπορεί να είναι πολύ βοηθητική. Συνοψίζοντας όλα αυτά, θα θέλαμε μια δομή που να αποθηκεύει στοιχεία συνδεδεμένα με μια αριθμητική τιμή, που να μπορεί να δώσει το στοιχείο με τη μικρότερη τιμή σε χρόνο τάξης $O(1)$, και με όλες τις απαραίτητες εντολές για ενημέρωση που διαχειρίζονται τη δομή να αποδίδουν παρόμοια, τουλάχιστον στο επίπεδο της διεπιφάνειας (interface level).

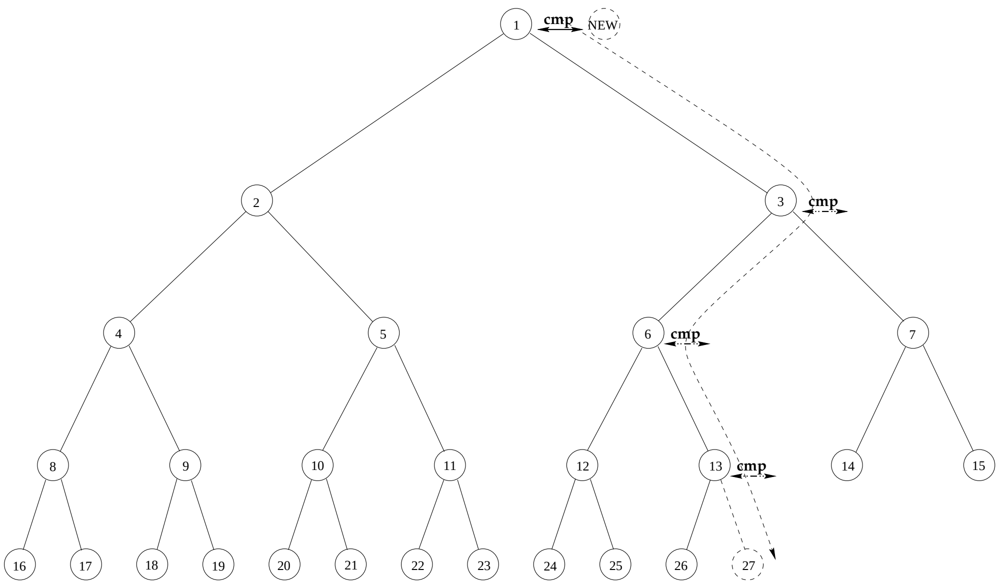
Η δομή που διαχειριζόμαστε είναι ένα δυαδικό δέντρο που κρατάει τα στοιχεία του με τέτοιο τρόπο ώστε να υλοποιεί μια ουρά προτεραιότητας. Ένα δέντρο που υλοποιεί μια ουρά προτεραιότητας είναι ένα δέντρο με κάθε του κόμβο να είναι μικρότερος ή ίσος από τους θυγατρικούς του κόμβους (όπως αναφέρθηκε ήδη στην ενότητα 2.2). Με αυτό τον τρόπο η ρίζα είναι το μικρότερο στοιχείο από όλα του δέντρου. Το δέντρο δεν ακολουθεί πλήρη διάταξη, ένα περιορισμός όμως που όπως αναφέρθηκε δεν είναι αναγκαίος. Όταν επιπλέον ασχοληθούμε με την αποδοτικότητα στη διαχείρηση μιας τέτοιας δομής, θέλουμε να έχει μία ακόμη ιδιότητα: να είναι ισορροπημένο και ακόμη περισσότερο πλήρως ισορροπημένο, δηλαδή να έχει όλα τα επίπεδα του δέντρου πλήρως κατηλειμένα, με μόνη πιθανή εξαίρεση το χαμηλότερο επίπεδο, που μπορεί να έχει τα αριστερότερα φύλλα του κατηλειμένα και τα δεξιότερα (αν υπάρχουν) να είναι ελεύθερα. Με αυτό τον τρόπο η απόσταση των φύλλων από τη ρίζα μπορεί να διαφέρει το πολύ κατά ένα, και καθόλου 'τρύπες' δεν εμφανίζονται στο δέντρο. Επομένως, στην ακόλουθη συζήτηση, όταν αναφερόμαστε στην *αναλοίωτη συνθήκη* του δέντρου, εννοούμε την ιδιότητα του ως ένα πλήρως ισορροπημένο, φίλτρο επιλογής ελαχίστου, δυαδικό δέντρο. Επιπλέον, η μερική διάταξη των στοιχείων που παρέχει ένα heap βοηθά στην αποδοτικότητα εκτέλεσης των ενημερώσεων.

Τώρα ας δούμε τι θα πρέπει να είναι το κάθε στοιχείο. Απο την ως τώρα συζήτηση, ξέρουμε ότι ένα στοιχείο θα πρέπει να αποτελείται από δύο αριθμούς: μία *ταυτότητα* (id) και την συσχετισμένη *προτεραιότητα* (priority). Το id μπορει να χρησιμοποιηθεί με πολλούς τρόπους, π.χ. σαν αριθμός ενός VC σε ένα δίκτυο ATM. Το priority είναι η τιμή που καθορίζει τη σχετική θέση του στοιχείου στο δέντρο. Είναι λοιπόν το μέτρο της διάταξης και της σύγκρισης μεταξύ των στοιχείων του δέντρου. Στο προηγούμενο παράδειγμα μπορεί να αναπαριστάνει την προτεραιότητα του αντίστοιχου VC. Εδώ θα θέλαμε να αναφέρουμε ότι στη συζήτηση που ακολουθεί και στις περιγραφές, όταν αναφερόμαστε σε ένα στοιχείο σαν μεγαλύτερο ή μικρότερο σε σχέση με κάποιο άλλο, αναφερόμαστε στη σύγκριση της αντίστοιχης τιμής προτεραιότητας. Το ίδιο ισχύει και σε κάθε αριθμητική αναφορά σε στοιχεία, εκτός αν σημειώνεται διαφορετικά.

4 Οι Λειτουργίες-Εντολές που Εκτελούνται σε μία Δενδρική Δομή και Μερικά Προβλήματα που Συναντήθηκαν

Εδώ θα παρουσιάσουμε τις λειτουργίες που θα μπορούσαν να υποστηρίζονται από ένα διαχειριστή σωρού (heap manager). Μια βασική παρουσίαση αυτών λειτουργιών και ο λόγος ύπαρξής τους δίνεται στο [Ioan00, section 4]. Εδώ δίνουμε μια γενική περιγραφή του πως αυτές διαχειρίζονται τη δενδρική δομή. Στη συνέχεια ασχολούμαστε με θέματα απόδοσης και εισάγουμε το pipelining. Ένα ειδικό πρόβλημα με το pipelining για τις εντολές delete παροουσιάζεται, μαζί με τη λύση του.

4.1 Μία Γενική Περιγραφή της Κάθε Λειτουργίας



Σχήμα 3: Μια σχηματική περιγραφή της εντολής Insert

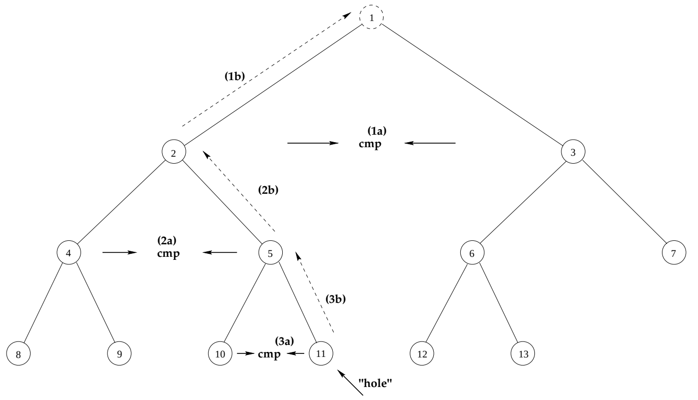
Εδώ θα δούμε πως η κάθε λειτουργία (εντολή) λαμβάνει χώρα στη δενδρική δομή, με όλες τις αναγκαίες ανατοποθετήσεις που απαιτούνται.

4.1.1 Η Διαχείρηση των Εντολών Insert από την Κορυφή Προς τα Κάτω (Top-to-Bottom)

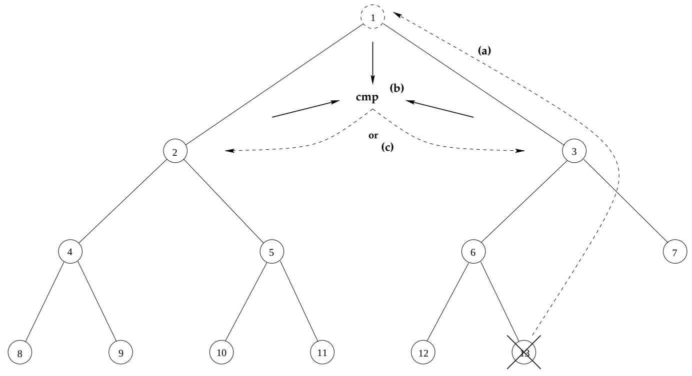
Ας ξεκινήσουμε τη συζήτησή μας με την εντολή insert. Η εντολή insert παίρνει σαν παράμετρο ένα νέο στοιχείο και πρέπει να το εισάγει στο δέντρο. Η insert διαισθητικά εκτελεί την εισαγωγή στο αριστερότερο άδειο φύλλο του κατώτερου επιπέδου του δέντρου, και μετά αναδιατάσει τα στοιχεία που βρίσκονται από πάνω, έτσι ώστε να διατηρήσει την ιδιότητα της ουράς προτεραιότητας σε ισχύ. Όμως, για να υποστηρίξουμε pipelining στη διαχείριση του heap, θα θέλαμε όλες οι εντολές να λειτουργούν top-to-bottom, και επομένως ξεκινούμε μια insert από τη ρίζα του δέντρου. Στην πραγματικότητα η insert δεν εκτελεί την εισαγωγή αμέσως. Το νεοεισαγώμενο στοιχείο θα πρέπει να βρει τη σωστή θέση του στο δέντρο. Σωστή με την έννοια του να διατηρεί η δενδρική δομή μας άθικτη την αναλοίωτη συνθήκη. Επομένως ένα νέο στοιχείο θα πρέπει σταδιακά να τοποθετηθεί στο αριστερότερο άδειο φύλλο του δέντρου (όχι αναγκαστικά περιέχοντας το νεοεισαχθέν στοιχείο), κρατώντας έτσι το δέντρο ισορροπημένο, και με κάθε κόμβο του να πραμένει μικρότερος από τους θυγατρικούς του κόμβους. Γι'αυτό μια insert συγκρίνει το νεοεισαγώμενο στοιχείο με το στοιχείο στη ρίζα, και τοποθετεί το μικρότερο εκ των δύο σαν τη νέα ρίζα (πιθανόν το ίδιο στοιχείο να συνεχίσει να είναι στη ρίζα). Αυτό συνεχίζει να γίνεται αναδρομικά, καθώς διατρέχουμε το δέντρο από την κορυφή προς τα κάτω. Το μονοπάτι που θα ακολουθήσουμε είναι γνωστό, μια και το αριστερότερο άδειο φύλλο μπορεί να εντοπιστεί λόγω της συνεχής μας γνώσης του συνολικού αριθμού των στοιχείων. Η εύρεση λοιπόν της διαδρομής είναι πλέον εύκολη, αλλά η υλοποίηση είναι κάπως περίπλοκη και παρουσιάζεται αργότερα.

4.1.2 Διαγραφές & Εναλλαγές

Ας ασχοληθούμε τώρα με την εντολή διαγραφής (delete). Η εντολή delete πρέπει να διαγράψει το ριζικό στοιχείο (επιστρέφοντας το περιεχόμενό του σαν αποτέλεσμα). Καθώς



Σχήμα 4: Η εντολή delete αφήνοντας μια 'τρύπα'



Σχήμα 5: Η εντολή delete αποφεύγοντας τη δημιουργία μιας 'τρύπας'

τώρα η ρίζα γίνεται άδεια, ένα άλλο στοιχείο του δέντρου πρέπει να τοποθετηθεί σαν μια νέα ρίζα. Αν επιλέξουμε έναν από τους θυγατρικούς κόμβους (το μικρότερο), τότε μία 'τρύπα' θα μετακινούνταν προς τα φύλλα του δέντρου, διαλέγοντας το μονοπάτι της 'τυχαία' καθώς διατρέχει τα επίπεδα του δέντρου. Εδώ το τυχαία δεν θα πρέπει να μεταφραστεί κυριολεκτικά. Αυτό που εννοούμε είναι ότι η κάθε καινούργια 'στροφή' επιλέγεται με το μέτρο της κατάταξης και το μονοπάτι που απαιτείται για να διατηρηθεί το δέντρο ισορροπημένο δεν ακολουθείται. Επομένως αυτή η μέθοδος δεν είναι η κατάλληλη. Αυτό που θα θέλαμε είναι να έχουμε κάθε νέα 'τρύπα' να 'τοποθετείται' στο δεξιότερο κατηλλειμένο φύλλο. Αυτό μπορεί να επιτευχθεί με το να διαγράψουμε το τελευταίο και να το μεταφέρουμε στη ρίζα, η οποία μόλις έγινε άδεια. Μετά η νεοεισαχθείσα ρίζα συγκρίνεται με τα παιδιά της και το μικρότερο από τα τρία στοιχεία εναλλάσεται με τη ρίζα. Αυτό γίνεται αναδρομικά για το τελευταίο ενναλαχθέν στοιχείο, εως ότου δεν χρειάζονται άλλες ανατοποθετήσεις. Σε αυτή τη στιγμή η λειτουργία της delete θεωρείται ως τελεσθείσα και το δέντρο παραμένει πλήρως ισορροπημένο και με την ιδιότητα της ουράς προτεραιότητας σε ισχύ.

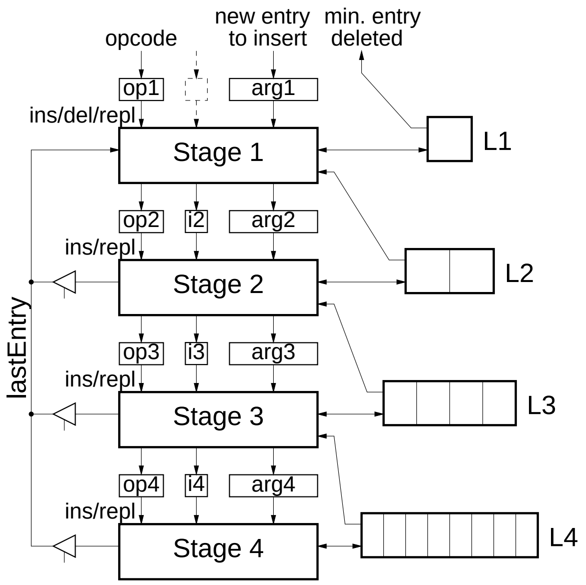
Τελικά συζητούμε την εντολή εναλλαγής (replace). Όπως αναφέρεται στο [Ioan00, section 4], αυτή η εντολή δεν υλοποιήθηκε στο σχέδιό μας, αλλά μπορεί να προστεθεί πολύ εύκολα, καθώς η υλοποίησή της μοιάζει πολύ με τον τρόπο που υλοποιήσαμε την εντολή delete. Η εντολή replace χρειάζεται μια τιμή σαν παράμετρο (θετική ή αρνητική) και προσθέτει αυτή την τιμή στο ριζικό στοιχείο (δηλ. αυξάνει τη ρίζα κατά την τιμή). Από εκεί και πέρα αυτή η εντολή δρά όπως η delete, αφότου η τελευταία έχει μετακινήσει το τελευταίο φύλλο στη ρίζα. Ένα νέο στοιχείο τποθετείται σαν νέα ρίζα και κάποια στοιχεία του δέντρου θα πρέπει να ανατοποθετηθούν με τη βοήθεια αναδρομικής σύγκρισης του μητρικού κόμβου με τα παιδιά του, ώστε να διατηρηθεί σε ισχύ η ιδιότητα της ουράς προτεραιότητας.

4.2 Εισάγωντας το Pipelining

Για να επιτύχουμε υψηλό throughput εντολών για τη δομή μας και το ακόλουθο σχέδιο/υλοποίηση, θα πρέπει να επιτρέπουμε νέες εντολές να εισέρχονται και να ξεκινούν να λειτουργούν (δηλ. να ανατοποθετούν τα στοιχεία του δέντρου) πριν η προηγούμενη να έχει τελειώσει. Με αυτό τον τρόπο, πολλές εντολές μπορούν να είναι σε εξέλιξη, λειτουργώντας στα στοιχεία του δέντρου. Για να επιτύχουμε τέτοιο παραλληλισμό, κάθε εντολή που έχει εισαχθεί θα πρέπει να διαχειρίζεται ένα ξεχωριστό σύνολο στοιχείων. Ο καλύτερος τρόπος να γίνει αυτό είναι να έχουμε όλες τις εντολές να διασχίζουν το δέντρο από την κορυφή προς τα κάτω με την πάροδο του χρόνου, και με την κάθε μία να ακολουθεί την προηγούμενή της με pipelined τρόπο. Αυτός είναι και ο λόγος που προσπαθήσαμε στις προηγούμενες συζητήσεις μας να έχουμε όλες τις εντολές να εξελίσονται από την κορυφή προς τα κάτω. Όμως, παρόλο που ο τρόπος αυτός μοιάζει πολύ αποδοτικός, πολύ πρόσθετη προσπάθεια χρειάζεται για το σχεδιασμό καθώς εξαρτήσεις δεδομένων (data hazards) παρουσιάζονται, και μονοπάτια προώθησης (forwarding paths) και λογική ελέγχου πρέπει να προστεθούν. Επιπλέον το να εισάγουμε κάθε νέα εντολή αμέσως μετά την προηγούμενή της (δηλ. στον επόμενο κύκλο ρολογιού) είναι πολύ δύσκολο (απαιτεί πολύ πρόσθετο υλικό) και το ίδιο ισχύει για την επίλυση κάποιων αντίστοιχων εξαρτήσεων δεδομένων. Αυτά συζητώνται αναλυτικά σε επόμενες ενότητες.

4.3 Αλγόριθμοι Σωρού για Pipelining

Το *Σχ. 6* παρουσιάζει τις βασικές ιδέες της διαχείρισης heap με pipelining. Κάθε επίπεδο του heap αποθηκεύεται σε ξεχωριστή φυσική μνήμη, και διαχειρίζεται από ένα αφοσιωμένο επίπεδο ελέγχου. Ο εξωτερικός κόσμος έρχεται σε άμεση επαφή μόνο με το επίπεδο 1. Οι εντολές που παρέχονται είναι: *(i) insert:* ένα νέο στοιχείο στο heap (στην άφιξη ενός πακέτου, όταν το flow παύει να είναι ανενεργό), *(ii) deleteMin:* διάβασμα και διαγραφή του ελάχιστου στοιχείου δηλ. της ρίζας (στην αναχώρηση πακέτου, όταν το flow γίνεται ανενεργό), και *(iii) replaceMin*[1]*:* αντικατέστησε το ελάχιστο με ένα νέο στοιχείο που έχει μεγαλύτερη τιμή (στην αναχώρηση πακέτου, όταν το flow παραμένει ενεργό).



Σχήμα 6: Απλοποιημένο διάγραμμα της pipeline

Όταν ζητείται από ένα επίπεδο να εκτελέσει μια εντολή, εκτελεί την εντολή στον κατάλληλο κόμβο του συγκεκριμένου επιπέδου, και μετά μπορεί να ζητήσει από το πιο κάτω επίπεδο να εκτελέσει μια γεννημένη εντολή που μπορεί να απαιτείται ώστε να διατηρηθεί η ιδιότητα του heap. Για τα επίπεδα 2 και πιο κάτω, εκτός από το να καθορίσουμε την εντολή και ένα δεδομένο, η αρίθμηση του κόμβου, i, θα πρέπει επίσης να καθοριστεί.Όταν οι εντολές του heap γίνονται με αυτό τον τρόπο, κάθε τμήμα (μαζί και το τμήμα 1 για I/O) είναι έτοιμο να εξυπηρετήσει μια νέα εντολή, από τη στιγμή που η προηγούμενη εντολή έχει ολοκληρώσει τη λειτουργία της στο δικό της επίπεδο μόνο.

Η εντολή *replace* είναι η ευκολότερη για κατανόηση. Στην *Σχ. 6* , το arg1 (παράμετρος 1) που δίνεται, πρέπει να αντικαταστήσει τη ρίζα στο επίπεδο 1. Το μέρος 1 (stage 1) διαβάζει τα δύο παιδιά του από το L2, για να αποφασίσει ποιά από τις τρεις τιμές είναι το νέο ελάχιστο, για να γραφτεί στο L1. Αν ένα εκ των δύο παιδιών ήταν το ελάχιστο, το δοσμένο arg1 θα πρέπει να αντικαταστήσει αυτό το παιδί, εκινώντας έτσι μια νέα εντολή *replace* για το μέρος 2, κ.ο.χ.

[1]μιλώντας εδώ θεωρητικά, αφου οι εντολές replace δεν υλοποιήθηκαν στο τελικό σχέδιο

Η εντολή *delete* είναι παρόμοια με τη replace. Για να διατηρήσουμε το heap ισορροπημένο, η ρίζα διαγράφεται αντικαθιστώντας τη με το δεξιότερο μη-άδειο στοιχείο στο κατώτερο μη-άδειο επίπεδο. Η αρτηρία (bus) lastEntry χρησιμοποιείται για να διαβάσουμε αυτό το στοιχείο, και το αντίστοιχο επίπεδο ενημερώνεται πως πρέπει να το διαγράψει. Όταν πολλές εντολές είναι σε εξέλιξη σε διαφορετικά μέρη της pipeline, το πραγματικό 'τελευταίο' στοιχείο μπορεί να μην είναι το τελευταίο στοιχείο του τελευταίου επιπέδου: η 'αιωρούμενη' τιμή του νεότερου-σε-εξέλιξη *insert* θα πρέπει να χρησιμοποιηθεί (περισσότερη ανάλυση δίνεται στο [Ioan00, section 4]). Σε αυτή την περίπτωση η αρτηρία lastEntry λειτουργεί ως ένα μονοπάτι προσπεράσματος (bypass path), και το πιο πρόσφατο insert ακυρώνεται.

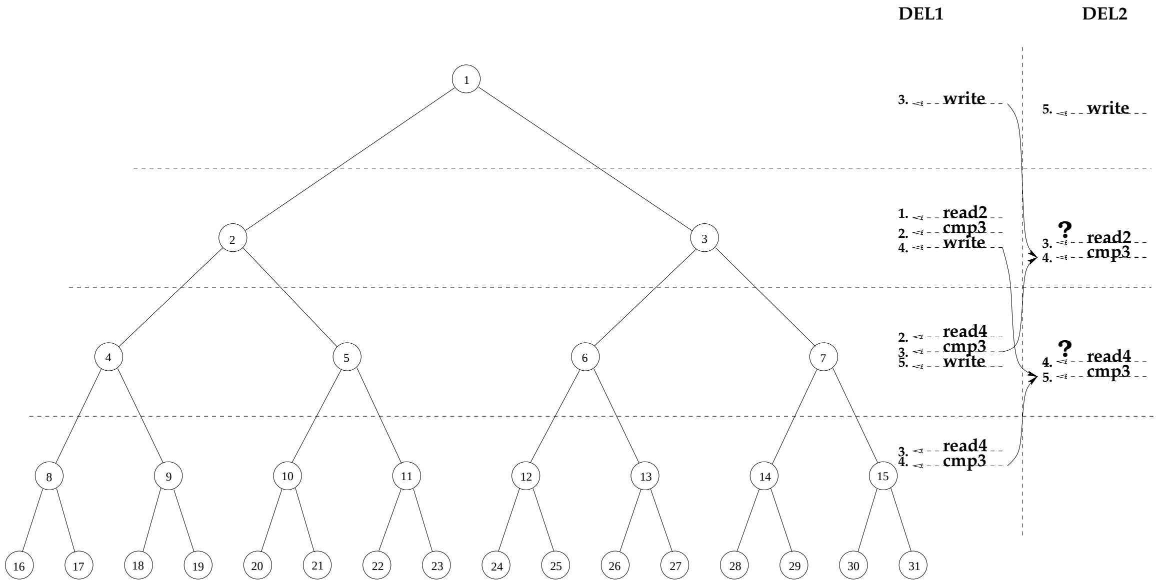
Ο παραδοσιακός αλγόριθμος για την *insert* θα πρέπει να τροποποιηθεί, όπως αναφέρεται στην ενότητα 4.1.1 . Σε ένα μη-pipelined heap, τα νέα στοιχεία εισάγονται στον 'πάτο' (bottom), μετά το τελευταίο μη-άδειο στοιχείο. Αν το νέο στοιχείο είναι μικρότερο από τον γονέα του, τότε εναλλάσσεται με αυτόν, κ.ο.κ. Τέτοιες εντολές θα εξελίσσονταν στη λάθος κατεύθυνση, στη pipeline του *Σχ. 6* . Στον τροποποιημένο αλγόριθμο, τα νέα στοιχεία εισάγονται στη ρίζα. Το νέο στοιχείο και η ρίζα συγκρίνονται. Το μικρότερο από τα δύο παραμένει ως η νέα ρίζα, και το άλλο αναδρομικά εισάγεται στο *κατάλληλο απο τα δύο* υπό-heaps. Με το να στρίβουμε σωστά –στο αριστερό ή δεξιό υπό-heap– αυτή την αλυσίδα των insert στο κάθε επίπεδο, μπορούμε να είμαστε βέβαιοι ότι το τελευταίο insert θα καθοδηγηθεί να γίνει ακριβώς στον κόμβο του heap δίπλα από το προηγούμενα-τελευταίο στοιχείο. Σημειώστε πως οι παραδοσιακές insert προχωρούν μόνο για όσα επίπεδα χρειάζεται, ενώ ο τροποποιημένος αλγόριθμός μας διατρέχει όλα τα επίπεδα. Αυτό πάντως δεν επηρεάζει το throughput των εντολών. Στο [Ioan00, section 4] μπορείτε να δείτε αναλυτικά κάποια από τα θέματα που αφορούν την αρχιτεκτονική.

5 Η Εξέλιξη των Ιδεών μας Μέσα από μια Σειρά Βημάτων και Κάποιες Ιδέες που Απορίφθησαν

Κάθε εντολή που εκτελείται στο δέντρο και ξεκινάει από τη ρίζα του, 'πέφτει' προς τα φύλλα με ταχύτητα ενός επιπέδου ανά έναν αριθμό κύκλων. Το κύριο θέμα για συζήτηση είναι η ταχύτητα της 'πτώσης' και επίσης ο ρυθμός εισαγωγής εντολών, δηλαδή το χρονικό διάστημα μετά το οποίο μία εντολή μπορεί να ακολουθήσει την προηγούμενή της. Αυτές οι παράμετροι καθορίζουν το throughput των εντολών που μπορεί να επιτευχθεί. Οι αρχικές ιδέες δίνονται στο [Ioan00, section 5]. Εδω βλέπουμε απευθείας τις τελικές λύσεις.

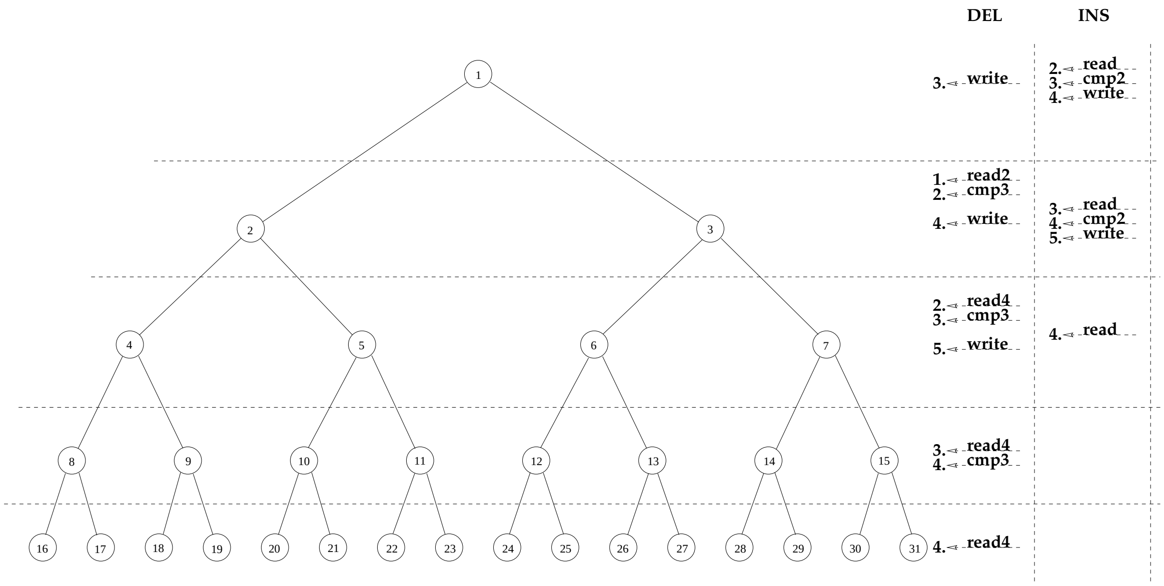
5.1 Χρησιμοποιώντας Όλους τους Δυνατούς Μηχανισμούς για Υψηλή Απόδοση

Σε αυτή την ενότητα παρουσιάζουμε την τελική μας απόφαση για τη διαχείριση της δομής, χρησιμοποιώντας όλους τους μηχανισμούς ([Ioan00, section 5]), και επιπλέον προσθέτοντας πιο 'επιθετική' προώθηση, καταλήγοντας σε ένα πολύ αποδοτικό σχέδιο. Τα διαγράμματα χρονισμού φαίνονται στα *Σχ. 7* , *Σχ. 8* , *Σχ. 9* και *Σχ. 10* .



Σχήμα 7: Ο χρονισμός για την ακολουθία DEL-DEL

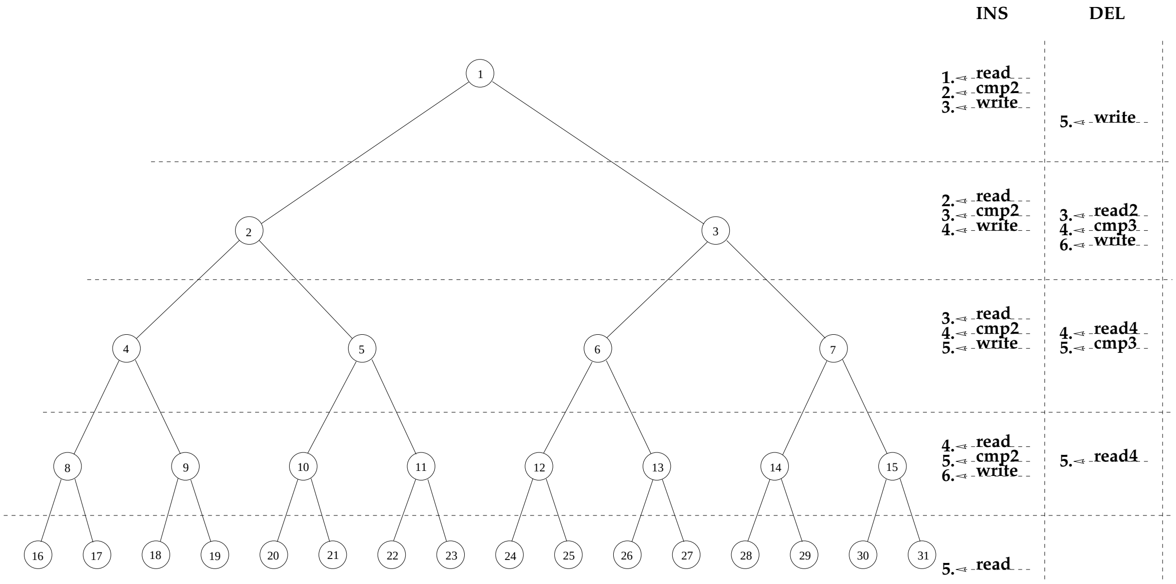
Θα ξεκινήσουμε με την εντολή insert που απεικονίζεται στο *Σχ. 10* , μια και φαίνεται πιο απλή. Σε αυτό το διάγραμμα βλέπουμε τη δεύτερη insert να εισάγεται αμέσως μετά την πρώτη στον κύκλο 2. Σε αυτή τη στιγμή η INS2 διαβάζει την τιμή που θέλει πριν ακόμη αυτή η τιμή αποφασιστεί από την σύγκριση στον κύκλο 2. Όμως αν είμαστε προσεκτικοί, αυτό μπορεί να δουλέψει σωστά, χωρίς να έχουμε καθυστερήσεις. Αυτό είναι αλήθεια, μια και η INS2 χρειάζεται την τιμή που διαβάζει, για την σύγκριση στον κύκλο 3. Σε αυτή την στιγμή η τιμή είναι γνωστή, μια και η σύγκριση της INS1 έχει γίνει στον κύκλο 1 και μπορεί να προωθηθεί. Όμως η ανάγνωση από τη μνήμη είναι χρήσιμη, γιατί καθώς διατρέχουμε το δέντρο προς τα κάτω, δύο συνεχόμενες εντολές insert μπορεί να πάρουν διαφορετική στροφή. Από εκεί και πέρα η τιμή που θα γραφτεί από την πρώτη είναι ανεξάρτητη από τη δεύτερη, και επομένως η δεύτερη insert μπορεί και πρέπει να διαβάσει το στοιχείο που θέλει, μόνη της. Η αμφιβολία του κατά πόσο η εντολή ανάγνωσης είναι χρήσιμη ή όχι, φαίνεραι στο διάγραμμα με ένα λατινικό ερωτηματικό πάνω από το βέλος της ανάγνωσης. Μία παρόμοια κατάσταση φαίνεται στο διάγραμμα του *Σχ. 8* , όπου μια insert που ακολουθεί μια delete εισάγεται και πάλι στον κύκλο 2, με την τιμή για τη σύγκριση στον κύκλο 3 να προωθείται από τη σύγκριση στον κύκλο 2.



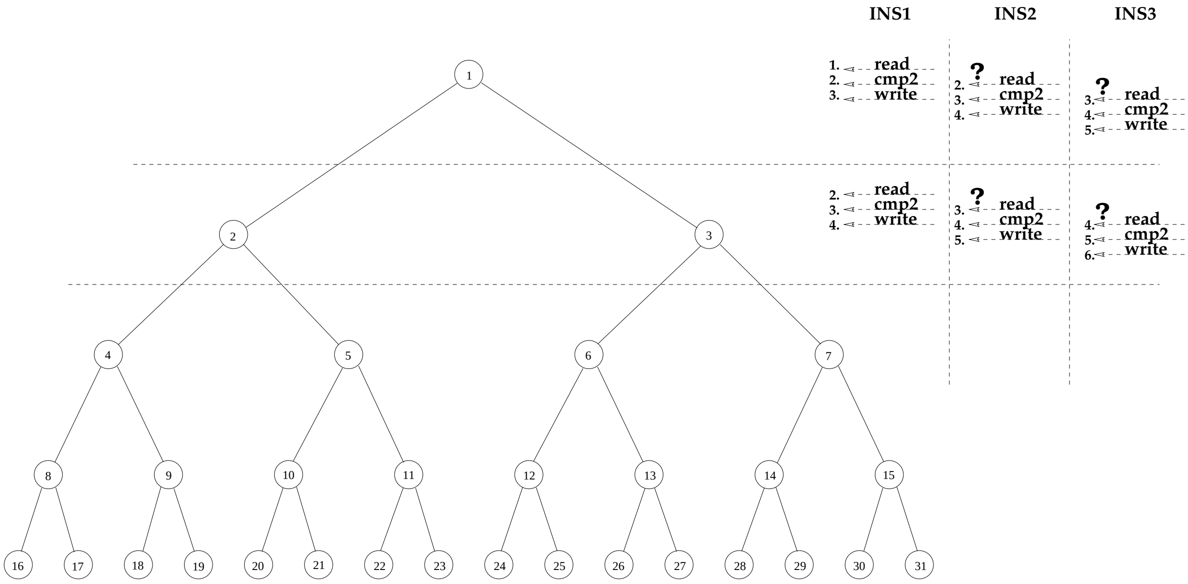
Σχήμα 8: Ο χρονισμός για την ακολουθία DEL-INS

Στο *Σχ. 7* μπορούμε να δούμε το χρονισμό για την εντολή delete. Η δεύτερη delete εισάγεται στον κύκλο 3. Επομένως η εντολή ανάγνωσης εκτελείται και πάλι πολύ νωρίς. Όμως η σύγκριση στον κύκλο 4 μπορεί και πάλι να έχει τις εισόδους της έγκυρες. Η ρίζα ήδη γράφεται στον κύκλο 3. Ένα από τα παιδιά πιθανώς γράφεται στον κύκλο 4 (αν χρειαζόταν να εναλλαχθεί με τη ρίζα), αλλά μπορεί να προωθηθεί μετά τη σύγκριση στον κύκλο 3. Το τρίτο στοιχείο (το δεύτερο παιδί) είχε σωστά διαβαστεί στον κύκλο 3 από την εντολή ανάγνωσης. Αν, μετά την κάθοδό μας για μερικά επίπεδα, τα δύο συνεχόμενα delete έχουν πάρει διαφορετική στροφή, τότε και τα δύο παιδιά θα παρθούν από την αντίστοιχη εντολή ανάγνωσης. Όμως για δύο συνεχόμενες delete, μια διαφορετική στροφή δεν τις κάνει ανεξάρτητες αμέσως, όπως γινόταν για τις insert. Μια και κάθε delete διαχειρίζεται ταυτόχρονα δύο επίπεδα του δέντρου, μία ακόμη 'πτώση' (δηλαδή ένας κύκλος) θα χρειαστεί για να γίνουν οι δύο ανεξάρτητες. Στο *Σχ. 9* βλέπουμε την περίπτωση όπου μια delete ακολουθεί μια insert. Η delete εισάγεται και πάλι στον κύκλο 3, και η ανάλυση ορθότητας μπορεί να γίνει με τον ίδιο τρόπο.

Μετά από αυτές τις βελτιώσεις έχουμε την κάθε insert να ακολουθεί την προηγούμενη εντολή στον αμέσως επόμενο κύκλο, ενώ η delete εισάγεται με ένα μόνο κύκλο κενό (idle). Αυτό δίνει ένα απλοποιημένο μέσο όρο για throughput εντολών, μιας ανα 1,5 κύκλων, με μόνο 0,5 κύκλους να μένουν idle.



Σχήμα 9: Ο χρονισμός για την ακολουθία INS-DEL



Σχήμα 10: Ο χρονισμός για την ακολουθία INS-INS

5.2 Πως ο Ρυθμός Εισαγωγής Μπορεί να Γίνει Ακόμη Πιο Υψηλός

Έχουμε λοιπόν πετύχει την απόδοση της ενότητας 5.1, όπου είπαμε πως όταν μια delete εισάγεται, θα πρέπει να προηγείται ένας κενός κύκλος. Επομένως όταν θέλουμε να εισάγουμε μια ακολουθία εντολών όπως INS-INS-DEL, θα πρέπει να εισαχθεί σαν INS-INS-idle-DEL. Όμως, για τους λόγους που φαίνονται στο [Ioan00, section 4], η νεότερη insert θα πρέπει να ακυρωθεί όταν εισαχθεί η delete. Μετά από αυτό η ακολουθία γίνεται INS-idle-idle-DEL. Οι δύο κενοί κύκλοι φαίνονται τώρα πολλοί, μια και ένας μόνο κενός κύκλος θα αρκούσε για να μη προκαλέσει προβλήματα στην pipeline. Αυτό μας δίνει την ιδέα να επιτρέψουμε μια DEL να ακολουθήσει αμέσως μια INS (με ένα κενό κύκλο να απαιτείται μόνο για τις ακολουθίες DEL-DEL, κάνοντας αυτές DEL-idle-DEL), μια και είμαστε σίγουροι πως η INS που προηγείται θα ακυρωθεί, προκαλώντας την αναγκαία φυσαλίδα (bubble) που θα διατρέξει την pipeline, μπροστά από την DEL. Επομένως με αυτή τη βελτίωση, έχουμε μια DEL να απαιτεί ένα κενό κύκλο μόνο για να διαχωριστεί εμέσως μετά από μια άλλη DEL. Ο υπολογισμός της προκύπτουσας συνολικής απόδοσης είναι κάπως παράξενος, μια και ένας χρονοδρομολογητής που έχει γνώση του μηχανισμού, θα μπορεί πιθανώς πάντα να χρονοδρομολογήσει μια ενδιάμεση INS για να διαχωρίσει μια ακολουθία DEL-DEL, μετατρέποντάς την σε DEL-INS-DEL. Η τελευταία μπορεί να εισαχθεί χωρίς πρόσθετους κύκλους, καταλήγοντας έτσι σε ρυθμό εισόδου μίας νέας εντολής ανά κύκλο ρολογιού. Όμως, αν υποθέσουμε έναν χρονοδρομολογητή που δεν είναι γνώστης, και με απλοποιημένα ποσοστά εντολών, μπορούμε να υποθέσουμε πως μια DEL θα ακολουθεί (σε πλήρη λειτουργία, δηλαδή όταν έχουμε σε κάθε κύκλο μια νέα εντολή) είτε μια DEL ή μια INS, με ίση πιθανότητα. Επομένως η πιθανότητα να έχουμε την ακολουθία DEL-DEL δίνεται από τον τύπο $P(DEL \cap DEL) = P(DEL) \times P(DEL) = 0.5 \times 0.5 = 0.25$. Επομένως ένας κενός κύκλος θα προστίθεται κάθε 4 εντολές (μετά από κάθε δεύτερη DEL) κατέ μέσο όρο, δίνοντας ταχύτητα 4 εντολών ανά 5 κύκλους.

Παρόλο που αυτό φαίνεται αρκετά απλό, ένα πρόβλημα προκύπτει, το οποίο ευτυχώς, αν γίνει αντιληπτό, μπορεί να ξεπεραστεί. Όταν μια DEL ακολουθεί αμέσως μια INS, είπαμε πως η INS θα ακυρωθεί, οδηγώντας στη δημιουργία μιας φυσαλίδας. Όμως, όπως καθορίζεται από τη σειριακή είσοδο των εντολών, η DEL θα πρέπει να έχει τα ίδια αποτελέσματα, όπως αν η INS είχε αφεθεί πρώτα να τελειώσει. Αυτό σημαίνει ότι η δομή της pipeline δεν θα πρέπει να επηρεάζει τα αποτελέσματα. Επομένως μια DEL, ασχέτως του κύκλου που εισάχθηκε, αν ακολουθεί τις ίδιες εντολές θα πρέπει να έχει τα ίδια αποτελέσματα, κρύβοντας έτσι τις λεπτομέρειες της υλοποίησης από τον έξω κόσμο. Όμως, όπως θα φανεί και στην ενότητα 6 , όταν η INS μιας ακολουθίας INS-DEL ακυρώνεται, δεν έχει ακόμη προλάβει να γράψει τη νέα ρίζα (αν χρειαζόταν σε περίπτωση που το νέο στοιχείο είναι το μικρότερο όλων). Επομένως η DEL θα δώσει μια (πιθανώς) παλιά ρίζα προς τα έξω, χρησιμοποιώντας το (πιθανώς μικρότερο) ακυρωμένο στοιχείο της INS σαν αντικαταστάτη της ρίζας. Αυτό το μικρότερο στοιχείο θα εξαχθεί από την επόμενη DEL, οδηγώντας στην εναλλαγή δυο συνεχόμενων τιμών (σε σύγκριση με τη σωστή αλολουθία εξόδου) στην ακολουθία εξόδου που προκύπτει από τον διαχειριστή του heap. Μια και αυτό συμβαίνει όταν η INS μιας ακολουθίας INS-DEL προσπαθεί να εισάγει ένα νέο ελάχιστο στοιχείο, μπορούμε να ξεπεράσουμε αυτό το πρόβλημα με το να έχουμε την DEL να λειτουργεί ελαφρώς διαφοροποιημένα. Δεν θα πρέπει αμέσως να εξάγει τη ρίζα, αλλά το μικρότερο της ρίζας και της ακυρωμένης τιμής. Έτσι πάντα θα εξάγει το ελάχιστο στοιχείο, αφήνοντας το μη επιλεγμένο για αντικαταστάτη της ρίζας (πιθανώς την ίδια την παλιά ρίζα, αλλά αυτό δεν προκαλεί κανένα επιπλέον πρόβλημα, αφού δε διαφέρει από τη γενική περίπτωση).

Ένας εναλλακτικός γραφικός τρόπος παρουσίασης και εξήγησης του χρονισμού δίνεται στο [Ioan00, section 5]. Οι δυσκολίες για την εισαγωγή μιας εντολής ανά κύκλο ρολογιού δίνονται επίσης στο [Ioan00, section 5] και η μελέτη περισσοτέρων εναλλακτικών κόστους και απόδοσης στο [Ioan00, section 6].

6 Η Υλοποίηση σε Υλικό (Hardware) του Πρωτοτύπου του Διαχειριστή Σωρού

Όπως είπαμε νωρίτερα, συνεχίσαμε την εργασία μας, προς την υλοποίηση ενός προτότυπου ASIC του ως τώρα διαχειριστή σωρού που περιγράψαμε. Θα περιγράψουμε τον όλο σχεδιασμό με μεγάλη λεπτομέρεια, ξεκινώντας με μια γενική εξωτερική περιγραφή που δίχνει το επίπεδο διεπιφάνειας. Μετά συνεχίζουμε παρουσιάζοντας τα σχετικά datapaths με την προσθήκη επιπλέον ανάλυσης για το πως λειτουργούν κάποια συγκεκριμένα τμήματα.

6.1 Η Διεπιφάνεια του Διαχειριστή Σωρού

Το περίγραμμα (outline) του διαχειριστή σωρού φαίνεται στο *Σχ. 11* . Ας ξεκινήσουμε εξηγώντας τα σήματα εισόδου και εξόδου. Ξεκινώντας με το σήμα *cmd*, είναι η εντολή που δίνεται στο διαχειριστή. Προς το παρόν, η insert και η delete είναι οι δύο δυνατές εντολές. Η είσοδος *valid* καθορίζει την εγκυρότητα της εντολής που δίνεται σε κάθε κύκλο. Η όλη διεπιφάνεια είναι συγχρονισμένη με την είσοδο ρολογιού *clk_in* και έτσι όλα τα σήματα δειγματοληπτούνται μία φορά σε κάθε περίοδο ρολογιού. Το *data_in* είναι τα δεδομένα

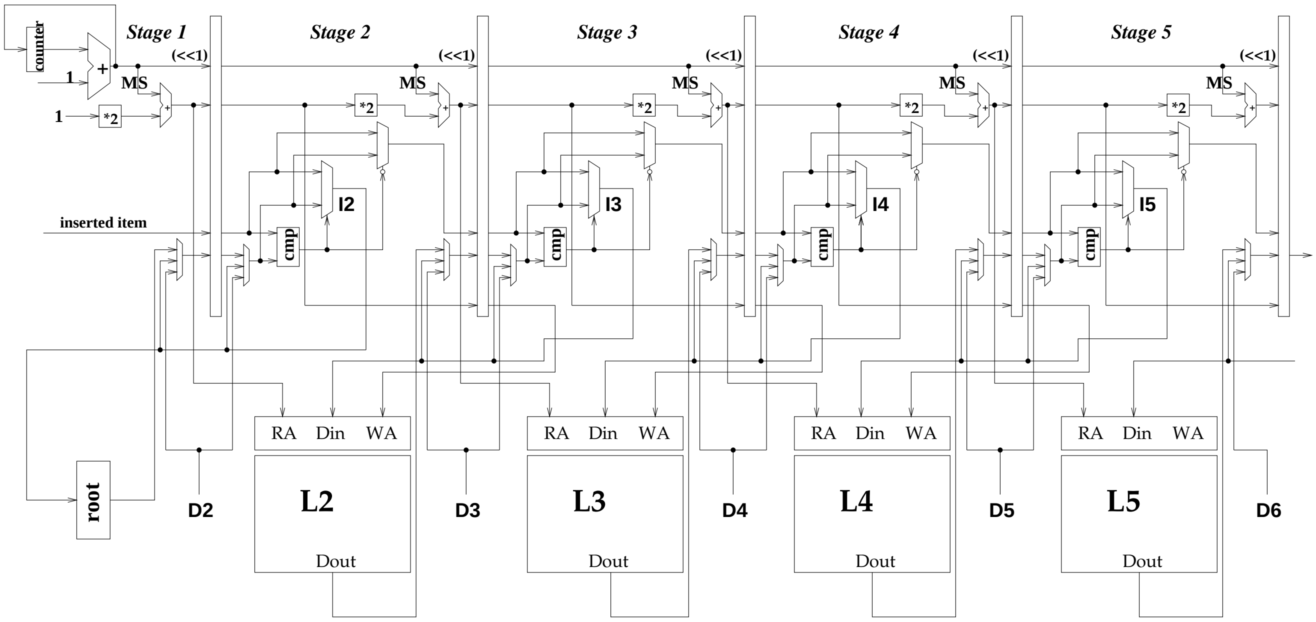
Σχήμα 11: Το περίγραμμα του διαχειριστή σωρού

που δίνονται σαν είσοδο, αν αυτά χρειάζονται. Αυτό οδηγείται στον ίδιο κύκλο μαζί με την αντίστοιχη εντολή. Η 'ταυτότητα' (id) και μια αντίστοιχη αριθμητική τιμή (π.χ. προτεραιότητα) χρειάζονται να δωθούν σε μια εντολή insert. Η delete δε χρειάζεται άλλες παραμέτρους εισόδου. Στην υλοποίησή μας το id είναι πλάτους 14 bits και δίνεται στα περισσότερο σημαντικά bits του *data_in*, ενώ η αντίστοιχη αριθμητική τιμή (ή απλά αριθμός για απλότητα) είναι πλάτους 18 bits και δίνεται στα λιγότερο σημαντικά bits του *data_in*. Το *reset* χρειάζεται για κάποιες απαραίτητες αρχικοποιήσεις στη έναρξη της λειτουργίας. Όσον αφορά τις εξόδους, το *reset_ack* σηματοδοτεί πότε η αρχικοποίηση ολοκληρώθηκε, οπότε η κανονική λειτουργία μπορεί να λάβει χώρα. Το *clk_out* είναι το σήμα ρολογιού που συγχρονίζει τα σήματα εξόδου. Τέλος, το *data_out* δίνει τα δεδομένα που επιστρέφονται από τον διαχειριστή. Μια insert δεν περιμένει δεδομένα εξόδου, ενώ η delete δέχεται το ελάχιστο στοιχείο μέσω του *data_out*. Το id και ο αριθμός περιέχονται στα 32 bits του *data_out* με τον ίδιο τρόπο που περιγράψαμε για το *data_in*. Ένα τελευταίο στοιχείο που θα θέλαμε να αναφέρουμε είναι πως οι εντολές insert μπορούν να εισαχθούν σε κάθε κύκλο, ενώ για μια delete θέλουμε ένα κενό κύκλο να τη διαχωρίσει από μια αμέσως προηγούμενη delete. Αυτό είναι το αποτέλεσμα αυτού που έχει ήδη εξηγηθεί στις ενότητες που περιγράψαμε την εκτέλεση με τρόπο pipelined και τις σχετικές μεθόδους για να επιτύχουμε υψηλό throughput εντολών.

6.2 Η Παρουσίαση των Datapaths

Το datapath του σχεδίου μας μπορεί να διαχωριστεί σε δύο σημαντικά μέρη. Το πρώτο περιλαμβάνει την αναγκαία λειτουργικότητα για τις εντολές insert, και το δεύτερο την αντίστοιχη λειτουργικότητα για τις εντολές delete. Ως συνήθως τα περισσότερα σήματα ελέγχου δεν φαίνονται. Η μνήμη φαίνεται και στα δύο datapaths, όμως υπάρχει μόνο ένα αντίτυπο που χρησιμοποιείται και από τα δύο (με πολύπλεξη των εισόδων και των εξόδων). Μια και οι δύο εντολές δεν μοιράζονται σχεδόν καθόλου από τη λειτουργία τους, ο διαχωρισμός των datapths ταιριάζει όμορφα και είναι βοηθητικός τόσο για την κατανόηση, αλλά και για την υλοποίηση του σχεδίου. Παρόλα ταύτα, τα δύο κατά τα άλλα ξεχωριστά datapaths επικοινωνούν μεταξύ τους μέσω σημάτων προσπέρασης και αντίστοιχων δεδομένων, τα οποία απαιτούνται για να υποστηριχθεί το πράγματι έντονο pipelining.

6.2.1 Το Μέρος του Datapath για τις Εντολές Insert

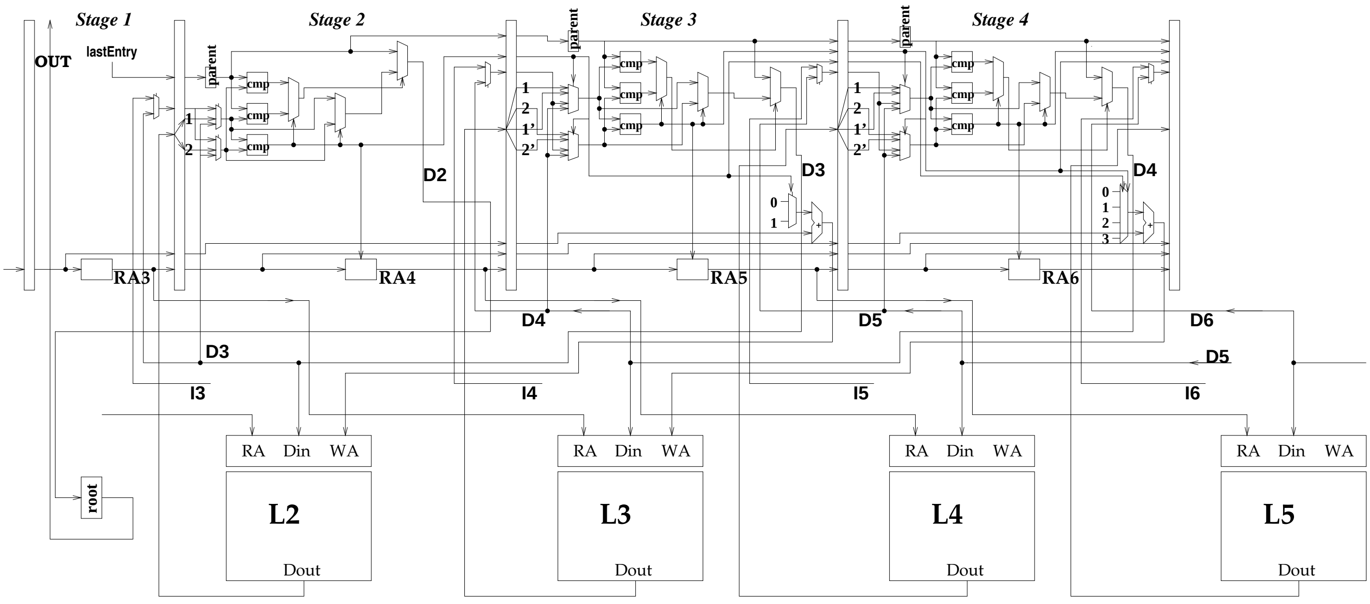


Σχήμα 12: Το μέρος του datapath για τις εντολές insert

Το datapath για την εντολή insert φαίνεται στο *Σχ. 12* . Με μια πρώτη ματιά παρατηρούμε πως είναι πολύ κανονικό (επαναλαμβανόμενο). Ένα τμήμα (stage) της pipeline καθιστά το βασικό τμήμα, το οποίο επαναλαμβάνεται ομοιόμορφα. Με αυτό τον τρόπο ο διαχειριστής του heap γίνεται εύκολα αναπροσαρμοζόμενου μεγέθους, απαιτώντας τον κατάλληλο αριθμό επαναλήψεων των τμημάτων ώστε να έχουμε τον διαχειριστή του κατάλληλου μεγέθους. Μόνο τα πρώτα δύο και το τελευταίο τμήμα διαφέρουν από το βασικό. Παρόλα αυτά, αποτελούν απλότερες μετατροπές του βασικού τμήματος. Το πρώτο τμήμα περιέχει επιπλέον λειτουργικότητα που αφορά τη διεπιφάνεια προς τον έξω κόσμο, αρχικούς υπολογισμούς και κάποια επιπλέον αποθήκευση για δεδομένα. Ξεκινώντας την περιγραφή μας από αυτό το τμήμα, βλέπουμε έναν μετρητή που κρατά τον αριθμό των στοιχείων που είναι ανά πάσα στιγμή αποθηκευμένα στη δομή. Κάθε φορά που μια insert (ή μια delete) εισάγεται, ο μετρητής αυξάνεται (ή μειώνεται). Μια περιγραφή σε μεγαλύτερο βάθος για τον μετρητή και το μηχανισμό ενημέρωσής του δίνεται στο [Ioan00, section 7], μια και κάποιες επιπλέον τεχνικές που απαιτούν περισσότερη ανάλυση χρησιμοποιούνται σε αυτό το μέρος. Στο τμήμα 1 η τιμή της ρίζας δίνεται στο επόμενο τμήμα. Η ρίζα κρατείται σε ένα καταχωρητή που αντιστοιχεί στο επίπεδο μνήμης 1. Το στοιχείο που θα εισαχθεί δίνεται επίσης στο τμήμα 2, αφού λήφθηκε μέσω της διεπιφάνειας. Ένα τελευταίο πράγμα που γίνεται σε αυτό το τμήμα είναι ο υπολογισμός της διεύθυνσης ανάγνωσης για το επόμενο επίπεδο. Αυτή η διεύθυνση υπολογίζεται με βάση την τιμή του μετρητή και την παρούσα διεύθυνση (δηλ. τη διεύθυνση που χρησιμοποιήθηκε για την ανάγνωση από τη μνήμη του αντίστοιχου τμήματος). Η τελευταία είναι 1, μια και αυτή η διεύθυνση αντιστοιχεί στη ρίζα. Όσο για τα σήματα ελέγχου που δε φαίνονται, πρέπει να αναφέρουμε πως ένα bit εγκυρότητας πρέπει να σχηματιστεί για να δίχνει ότι μια εντολή insert έχει εισαχθεί. Αυτό θα δίνεται και στα επόμενα τμήματα και θα συμμετάσχει στον υπολογισμό των σημάτων oe, we και cs.

Στη συνέχεια περιγράφουμε το βασικό τμήμα, το οποίο, χρησιμοποιούμενο τον κατάλληλο αριθμό φορών, δημιουργεί έναν διαχειριστή του απαιτούμενου μεγέθους. Δε θα εξηγήσουμε ξεχωριστά το δεύτερο και το τελευταίο τμήμα, τα οποία αναφέραμε πως διαφέρουν από το βασικό, μια και αποτελούν απλούστερες διαφοροποιήσεις του τελευταίου, εύκολα κατανοητές. Καθώς δίνουμε την περιγραφή, ο αναγνώστης μπορεί να παραπέμπεται στο τμήμα 3 (stage 3) του *Σχ. 12* . Πρώτα απ'όλα βλεπουμε στα αριστερά ένα συγκριτή (cmp) με δύο εισόδους. Η πρώτη (πάνω) είναι το στοιχείο που 'αιωρείται' προς τον πάτο του δέντρου, ενώ το δεύτερο είναι το στοιχείο που είναι προς το παρόν αποθηκευμένο στο προηγούμενο επίπεδο του δέντρου. Το μονοπάτι που ακολουθείται καθορίζεται από το μονοπάτι που πρέπει να ακολουθήσουμε για να φτάσουμε στην κενή θέση του δέντρου που θα φιλοξενήσει το νέο στοιχείο. Όμως, καθώς άλλες εντολές μπορεί να διασχίζουν το δέντρο, αυτό το στοιχείο, που διαισθητικά καταλαμβάνει τη θέση στο δέντρο, ίσως δεν είναι εφικτό να δωθεί από τη μνήμη του επιπέδου 2, μια και αυτή μπορεί να περιέχει παλιά δεδομένα. Ένα μονοπάτι προσπέρασης πρέπει να ενεργοποιηθεί για να προσφέρει τα δεδομένα που αυτή τη στιγμή προσπαθούν να εγγραφούν στη μνήμη (όλες οι προσπεράσεις αναλύονται στο [Ioan00, section 7]). Δύο πολυπλέκτες, ένας στα αριστερά του τμήματος 3 και ένας στα δεξιά του τμήματος 2, επιλέγουν την κατάλληλη είσοδο. Αναφερόμενοι στις εισόδους από τον πρώτο (τμήμα 2) πολυπλέκτη στο δεύτερο, και παίρνοντας τις εισόδους του κάθε πολυπλέκτη από πάνω προς τα κάτω, έχουμε τις εξής περιπτώσεις: (α) Παίρνουμε τα δεδομένα απευθείας από τη μνήμη. Αυτό συμβαίνει όταν καμία προηγουμένως εισηγμένη εντολή που ενέχει εγγραφή στο επίπεδο 2 δεν εκτελείται αυτή τη στιγμή. (β) Μια insert είχε προηγουμένως εισαχθεί με έναν ενδιάμεσο κενό κύκλο. (γ) Μια delete είχε προηγουμένως εισαχθεί με έναν ενδιάμεσο κενό κύκλο. (δ) Αυτή είναι η είσοδος που λαμβάνεται από τον πρώτο πολυπλέκτη. (ε) Μια insert είχε εισαχθεί στον προηγούμενο κύκλο. (στ) Μια delete είχε εισαχθεί στον προηγούμενο κύκλο. Αφότου έχουμε καθορίσει και τα δύο στοιχεία, ο συγκριτής βρίσκει αυτό με τη μικρότερη τιμή προτεραιότητας. Αυτό επιλέγεται από τον πρώτο πολυπλέκτη για να γραφτεί στη μνήμη L2, στη διεύθυνση από την οποία προσπαθήσαμε προηγουμένως να διαβάσουμε. Το άλλο στοιχείο δίνεται μέσω του άλλου πολυπλέκτη στο επόμενο τμήμα και καθιστά το επωνομαζόμενο 'αιωρούμενο' αντικείμενο. Η νέα διεύθυνση για ανάγνωση υπολογίζεται επίσης. Αυτό που φαίνεται σαν ένας πολλαπλασιαστής (*2) και ένας αθροιστής, υλοποιούνται με απλή διαχείριση των bits.

6.2.2 Το Μέρος του Datapath για τις Εντολές Delete



Σχήμα 13: The datapath portion for the delete operation

Το datapath για τις εντολές delete φαίνεται στο *Σχ. 13* . Και αυτο το datapath είναι πολύ κανονικό, αν και αυτό δεν μπορεί να φανεί καλά στο σχήμα. Αυτό συμβαίνει γιατί το βασικό τμήμα χρησιμοποιείται από το τμήμα 4 και πέρα. Επομένως έχουμε τα πρώτα 3 να αποτελούν απλότερες μετατροπές του βασικού τμήματος. Επαναλαμβάνοντας το τελευταίο μπορούμε να έχουμε τον διαχειριστή του κατάλληλου μεγέθους. Το τελευταίο τμήμα αποτελεί και πάλι απλούστευση του βασικού. Το πρώτο τμήμα είναι η απλούστερη όλων των μετατροπών, αλλά περιλαμβάνει επιπλέον λειτουργικότητα για τους λόγους που αναφέρθηκαν και στο πρώτο επίπεδο της insert. Διαβάζει τη ρίζα και εξάγει το ελάχιστο στοιχείο, και η ρίζα στη συνέχεια θα αντικατασταθεί. Το νέο στοιχείο που (διαισθητικά) θα τοποθετηθεί προσωρινά στη ρίζα, όπως περιγράφτηκε στην 4.1, θα δωθεί είτε από μια ακυρωμένη insert, ή απευθείας από τη θέση της μνήμης που βρίσκεται. Αυτό φαίνεται απλά με ένα βέλος στο σχήμα. Η διεύθυνση ανάγνωσης για τη μνήμη L3 επίσης υπολογίζεται.

Ας περιγράψουμε τώρα τη λογική για το τμήμα 4 του *Σχ. 13* . Αυτό είναι ένα αντίγραφο του βασικού τμήματος για τη delete, και μέσω αυτού μπορούμε να καταλάβουμε πως λειτουργούν και τα άλλα τμήματα που αναφέρθηκαν σαν απλούστερες μετατροπές. Τέσσερα στοιχεία δίνονται από το προηγούμενο τμήμα, τα οποία σημειώνονται σαν 1, 2, 1' και 2'. Αυτά είναι τα τέσσερα παιδιά που χρειάζεται να διαβαστούν από το προηγούμενο τμήμα για τους λόγους αποδοτικότητας που αναφέρθηκαν όταν συζητούσαμε πως εκτελούνται οι εντολές delete (δείτε επίσης το [Ioan00, section 5]). Όμως την ώρα που φτάνουμε το τμήμα 4, ξέρουμε ποιά δύο παιδιά είναι τα χρήσιμα. Επιπλέον, μονοπάτια προσπεράσματος είναι υποψήφια να δώσουν τις πραγματικές τιμές των παιδιών, μια και η μνήμη μπορεί να περιέχει παλιά δεδομένα. Έχουμε τρεις περιπτώσεις που χρειάζονται ένα μονοπάτι προσπεράσματος να ενεργοποιηθεί, οι οποίες αντιστοιχούν σε συγκεκριμένες ακολουθίες εντολών. (α) Μια DEL είχε εισαχθεί μέσω της ακολουθίας INS-INS-idle-DEL. Η δεύτερη insert θα έχει ακυρωθεί για λογαριασμό της DEL, όμως η πρώτη θα είναι ακόμη ενεργή και προσπαθεί να γράψει το επίπεδο της μνήμης από το οποίο προσπαθούμε να διαβάσουμε. (β) Μια DEL ακολουθεί την παρούσα DEL με δυο ενδιάμεσους κενούς κύκλους. (γ) Μια DEL ακολουθεί την παρούσα DEL με ένα μόνο ενδιάμεσο κύκλο. Οι αντίστοιχοι πολυπλέκτες που αποφασίζουν ποιο μονοπάτι θα ενεργοποιηθεί είναι ο 2-σε-1 πολυπλέκτης στο δεξί πάνω άκρο του τμήματος 3 και ο 4-σε-1 πολυπλέκτης στα αριστερά του τμήματος 4. Οι περιπτώσεις προσπεράσματος που αναφέρθηκαν πιο πάνω ακολουθούν τη σειρά εισόδων των πολυπλεκτών από τον αριστερό (τμήμα 3) στον δεξιό και από πάνω προς τα κάτω. Έχοντας αποφασίσει τα δύο παιδιά που θα χρησιμοποιηθούν, αυτά συγκρίνονται μεταξύ τους και με το γονέα τους για να καθοριστεί το μικρότερο από τα τρία. Ο πρώτος πολυπλέκτης (ματά τους συγκριτές) διαλέγει το αποτέλεσμα της σύγκρισης ανάμεσα στο γονέα και το μικρότερο παιδί. Το μικρότερο παιδί προωθείται μέσω του επόμενου πολυπλέκτη και ο τρίτος πολυπλέκτης επιλέγει το μικρότερο μεταξύ του γονέα και του μικρότερου παιδιού. Το στοιχείο που επιλέγεται θα γραφτεί στην κατάλληλη θέση της μνήμης. Μπορούμε να υπολογίσουμε την διεύθυνση αυτή μια και ξέρουμε τη διεύθυνση που χρησιμοποιήθηκε για να διαβάσουμε τα 4 παιδιά και επιπλέον ξέρουμε τα αποτελέσματα των συγκρίσεων από τα προηγούμενα τμήματα. Ο πολυπλέκτης στο σχήμα που επιλέγει μεταξύ των 0, 1, 2 ή 3 και ο αθροιστής υλοποιούνται με απλή διαχείριση των bits. Τέλος βλέπουμε κάτω τον υπολογισμό της νέας διεύθυνσης ανάγνωσης. Κι αυτή επίσης απαιτεί απλή διαχείριση των bits.

Διάφορα θέματα που επέφεραν δυσκολίες στην αρχιτεκτονική δίνονται στο [Ioan00, section 7] μαζί με τις λύσεις τους.

6.3 Επαλήθευση της Ορθής Λειτουργίας

Για να επαληθεύσουμε το σχέδιό μας, συγγράψαμε τρία μοντέλα ενός heap, με διαφορετικό επίπεδο αφαίρεσης, και τα προσομοιώσαμε παράλληλα με το σχέδιο σε Verilog, έτσι ώστε κάθε υψηλότερου επιπέδου μοντέλο να ελέγχει την ορθότητα του επόμενου (πιο κάτω) επιπέδου, ως το τελικό επίπεδο-σχέδιο. Το υψηλότερου επιπέδου μοντέλο, γραμμένο σε Perl, είναι μια ουρά προτεραιότητας που απλά επαληθεύει οτι το στοιχείο που επιστρέφεται από τις delete είναι κάθε φορά το ελάχιστο όσων έχουν εισαχθεί και δεν έχουν διαγραφεί. Το επόμενο πιο αναλυτικό μοντέλο, γραμμένο σε C, είναι ένα απλό heap. Τα περιεχόμενα της μνήμης του πρέπει να ταιριάζουν με αυτά της Verilog για σχήματα ελέγχου που δεν ενεργοποιούν το μηχανισμό ακύρωσης των insert ([Ioan00, section 4]). Όμως, όταν αυτός ο μηχανισμός ενεργοποιείται, η συνολική τοποθέτηση των στοιχείων στο pipelined heap μπορεί να διαφέρει από αυτή του απλού heap, γιατί μερικές insert ακυρώνονται πρίν φτάσουν το τελειωτικό τους στάδιο, και επομένως η τιμή που θα αντικαταστήσει τη ρίζα στην επόμενη delete μπορεί να μην είναι η μέγιστη του μονοπατιού της insert (όπως συνέβαινε στο απλό heap), αλλά μια άλλη τιμή του μονοπατιού, όπως καθορίζεται από το σχετικό χρονισμό των εντολών. Το πιο αναλυτικό μοντέλο C αναλυτικά περιγράφει-προσομοιώνει αυτή τη συμπεριφορα. Έχουμε επαληθεύσει το σχέδιο με πολλές διαφορετικές ακολουθίες εντολών, ενεργοποιώντας όλα τα πιθανά μονοπάτια προσπεράσματος. Σχήματα ελέγχου (test patterns) δεκάδων χιλιάδων εντολών χρησιμοποιήθηκαν, ώστε να ελεγχθούν όλα τα επίπεδα του heap, φτάνοντας και σε συνθήκες κορεσμού. Διάφορες παράμετροι μπορούν να δωθούν, όπως ο μέγιστος αριθμός πληρότητας που θα επέλθει, τα ήδη των ακολουθιών εντολών (όπως π.χ. τυχαία, ή συνεχόμενες insert ακολουθούμενες από συνεχόμενες delete για τη συνεχή ενεργοποίηση του μηχανισμού της ακύρωσης των insert κλπ.), ή ο ελάχιστος αριθμός στοιχείων που μπορεί να παραμείνει στο δέντρο (κρατώντας το έτσι αρκετά βαθύ όταν θέλουμε, ώστε να ενεργοποιούνται πολύπλοκοι μηχανισμοί, ή να επιτρέπεται στο heap να αδειάζει, ώστε να ελεγχθεί η συμπεριφορά του σε μια τέτοια περίπτωση). Ο αριθμός id (π.χ. VC) για κάθε στοιχείο που εισάγεται, δίνεται από ένα σύνολο VCs. Αυτό το σύνολο ενημερώνεται με κάθε εντολή που εισάγεται. Για να ξέρουμε ποιο id θα ελευθερωθεί από μια delete, η γεννήτρια των ακολουθιών έχει γνώση του τρόπου που λειτουργεί το heap (με τη βοήθεια του απλού μοντέλου Perl). Επίσης, οι προτεραιότητες δίνονται με τρόπο που να ικανοποιεί τις ανάγκες του μηχανισμού σύγκρισης που δίνεται στο [Ioan00, section 7], οπότε η γεννήτρια έχει γνώση του μηχανισμού επανατύλιξης (wrap-around). Τέλος scripts σε Perl χρησιμοποιούνται για να επαληθεύσουν την ορθή λειτουργία του σχεδίου σε Verilog, συγκρινόμενο με τα μοντέλα επαλήθευσης.

6.4 Τα Αποτελέσματα σε Κόστος και Απόδοση

Όπως έχουμε αναφέρει πριν, η υλοποίησή μας αποτελείται από στοιχεία που έχουν μέγεθος 32 bits, και το μεγαλύτερο κομμάτι SRAM είναι 2Kx32, με κύκλο ρολογιού περίπου στα 10ns. Μιλώντας με ορολογία Verilog, το datapath του βασικού τμήματος της pipeline έχει πολυπλοκότητα περίπου 400 bits καταχωρητών, 160 bits πολυπλεκτών δύο εισόδων, 220 bits πολυπλεκτών τεσσάρων εισόδων, και 100 bits αριθμητικής σύγκρισης. Το τμήμα ελέγχου (περιλαμβανομένου και του ελέγχου του προσπεράσματος) έχει πολυπλοκότητα περίπου 50 γραμμών σε κώδικα Verilog.

Επεξεργαστήκαμε το σχέδιο μέσω του Synopsys και καταλήξαμε στα παρακάτω αποτελέσματα. Ξεκινώντας με τη διάσταση του σχεδίου, η μη συνδιαστική λογική καταλαμβάνει χώρο $2.72mm^2$ και η συνδιαστική $67mm^2$. Από τα τελευταία $67mm^2$, όπως φαίνεται στο [Ioan00, section 7], τα $55mm^2$ καταλαμβάνονται από τη μνήμη που αποθηκεύει τα στοιχεία του δέντρου. Τα υπόλοιπα αφορούν καταχωρητές τις pipeline και άλλους καταχωρητές του datapath. Η συνολική επιφάνεια είναι λοιπόν $69.72mm^2$, χωρίς την επιφάνεια της διασύνδεσης. Αυτή η επιφάνεια προσθέτει περίπου 15%, κάνοντας το τελικό σχέδιο συνολικής επιφάνειας $80mm^2$.

Ο κύκλος ρολογιού είναι 17 ns για συνθήκες worst case. Το μονοπάτι που δίνει τη μαγαλύτερη καθυστέρηση είναι αυτό που βρίσκει το τελυταίο στοιχείο του heap, όταν αυτό βρίσκεται στην τελευταία μνήμη (L14), μια και αυτή είναι η μεγαλύτερη και η αργότερη. Αυτός ο κύκλος οδηγεί σε ένα ρολόι περίπου 60 MHz, δίνοντας ρυθμό εντολών 60 Mops.

Συμπεράσματα

Έχουμε παρουσιάσει μια σύνοψη προηγμένων αλγόριθμων χρονοδρομολόγησης. Όπως είδαμε, η εύρεση του ελάχιστου από ένα μεγάλο σύνολο τιμών είναι η κοινή υπολογιστική ανάγκη. Η δομή δεδομένων του σωρού (heap) παρουσιάστηκε στη συνέχεια σαν μια ελπιδοφόρα βάση για την επίλυση αυτού του προβλήματος. Οι τροποποιήσεις που χρειάζονται στους παραδοσιακούς αλγόριθμους για heap παρουσιάστηκαν στη συνέχεια. Συνεχίζοντας την εργασία μας, υλοποιήσαμε έναν διαχειριστή σωρού (heap manager) που χρησιμοποιεί pipelining, δίχνοντας έτσι την εφικτότητα μεγάλων ουρών προτεραιότητας, με ρυθμούς της τάξης των 100 εκ. εντολών το δευτ. με ρυθμούς ρολογιού της τάξης λίγων εκατοντάδων ρολογιού, με μοντέρνα τεχνολογία. Το κόστος τέτοιων heap managers είναι η (αναπόφευκτη) μνήμη που κρατάει τις τιμές προτεραιότητας και τις ταυτότητες των ροών, και επίσης μια ντουζίνα ή περίπου τόσα τμήματα της pipeline όχι ιδιαίτερης πολυπλοκότητας.

Αυτό συγκρίνεται αρκετά ευνοϊκά με τα calendar queues – την εναλλακτική υλοποίηση για υλοποιήσεις ουρών – με την αυξημένη τους ανάγκη σε μνήμη και την πολυπλοκότητα να διαχειρίζονται συγκρούσεις. Έχουμε παρουσιάσει σε μεγάλη λεπτομέρεια τον τρόπο που ο heap manager λειτουργεί, δίνοντας λύσεις σε πολλά προβλήματα που προκύπτουν όταν εισάγουμε το pipelining. Επιπλέον παρουσιάσαμε ένα σύνολο εναλλακτικών λύσεων που ανταλλάσουν απόδοση και κόστος. Αυτά υποδικνύουν και το λόγο που η συγκεκριμένη υλοποίηση ακολουθήθηκε. Μπορεί επίσης να αποτελέσει τη βάση για λύσεις χαμηλότερου κόστους, αν κάποιος ενδιαφέρεται για μια τέτοια υλοποίηση. Τέλος πολλές από τις λύσεις που δόθηκαν σε προβλήματα μπορούν να χρησιμοποιηθούν ξεχωριστά, αν ανάλογα προβλήματα εμφανιστούν, σε πιθανώς μη σχετικές υλοποιήσεις.

Η εφικτότητα ουρών προτεραιότητας με πολλές χιλιάδες στοιχεία στο εύρος των 100 Mops αφορά πολύ τα δίκτυα υψηλών ταχυτήτων και το μελλοντικό διαδίκτυο. Οι περισσότεροι από τους προηγμένους αλγόριθμους για την παροχή εγγυήσεων εξυπηρέτησης υψηλού επιπέδου, βασίζονται σε ουρές ανά ροή και σε χρονοδρομολογητές βασισμένους σε ουρές προτεραιότητας. Επομένως, επιδικνύουμε ότι τέτοιοι αλγόριθμοι είναι εφικτοί, σε χαμηλό κόστος και για ρυθμούς OC-192 (10 Gbps) και υψηλότερους.

Παραρτήματα

A Χαρακτηριστικά Μνημών

	AREA mm^2	**ACCESS TIME(ns)**	**DATA SETUP(ns)**	**ADDR SETUP(ns)**	**CYCLE TIME(ns)**	**mux**
32 x 16	0.385	3.38	2.58	1.92	6.76	4
64 x 32	0.754	3.53	2.67	1.98	7.03	4
128 x 32	0.895	3.44	2.86	2.09	6.86	4
256 x 32	1.162	3.54	2.84	2.14	7.14	4
512 x 32	1.712	3.62	2.92	2.30	7.53	4
1024 x 32	2.818	3.84	2.94	2.55	8.49	4
2048 x 32	5.052	4.51	3.54	2.51	9.86	8

Table 1: Διάφορες μνήμες και τα χαρακτηριστικά τους.

B Πληροφορίες Σχετικά με τον Κώδικα

Κώδικας Verilog	3200 γραμμές
Αυτοματοποιημένος Κώδικας Verilog	3100 γραμμές
Συνολικός Κώδικας (χωρίς μνήμες)	6300 γραμμές
Κώδικας Perl	450 γραμμές
Κώδικας C	550 γραμμές
Μεγέθη Ελέγχου	ως και 100K εντολές
Scripts για το Synopsys	250 γραμμές

Βιβλιογραφία

[Rexford96] J.L. Rexford, A.G. Greenberg and F.G. Bonomi, *"Hardware-Efficient Fair Queueing Architectures for High Speed Networks"*, INFOCOM '96, pp 638-646

[Tarjan83] Robert Endre Tarjan, *"Data Structures and Network Algorithms"*, (CBMS-NSF Regional Conference Series in Applied Mathematics), 1983, ISBN 0-89871-187-8.

[Bennett97] J. Bennett, H. Zhang: "Hierarchical Packet Fair Queueing Algorithms", *IEEE/ACM Trans. on Networking,* vol. 5, no. 5, Oct. 1997, pp. 675-689.

[Brown88] R. Brown: "Calendar Queues: a Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem", *Commun. of the ACM,* vol. 31, no. 10, Oct. 1988, pp. 1220-1227.

[Chao91] H. J. Chao: "A Novel Architecture for Queue Management in the ATM Network", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 9, no. 7, Sep. 1991, pp. 1110-1118.

[Chao97] H. J. Chao, H. Cheng, Y. Jeng, D. Jeong: "Design of a Generalized Priority Queue Manager for ATM Switches", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 15, no. 5, June 1997, pp. 867-880.

[Chao99] H. J. Chao, Y. Jeng, X. Guo, C. Lam: "Design of Packet-Fair Queueing Schedulers using a RAM-based Searching Engine", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 17, no. 6, June 1999, pp. 1105-1126.

[Jones86] D. Jones: "An Empirical Comparison of Priority-Queue and Event-Set Implementations", *Commun. of the ACM,* vol. 29, no. 4, Apr. 1986, pp. 300-311.

[Kate87] M. Katevenis: "Fast Switching and Fair Control of Congested Flow in Broad-Band Networks", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 5, no. 8, Oct. 1987, pp. 1315-1326.

[Kate97] M. Katevenis, lectures on heap management, Fall 1997.

[KaSC91] M. Katevenis, S. Sidiropoulos, C. Courcoubetis: "Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 9, no. 8, Oct. 1991, pp. 1265-1279.

[KaSM97] M. Katevenis, D. Serpanos, E. Markatos: “Multi-Queue Management and Scheduling for Improved QoS in Communication Networks”, *Proceedings of EMMSEC'97* (European Multimedia Microprocessor Systems and Electronic Commerce Conference), Florence, Italy, Nov. 1997, pp. 906-913; http://archvlsi.ics.forth.gr/html_papers/EMMSEC97/paper.html

[Keshav97] S. Keshav: ”An Engineering Approach to Computer Networking”, *Addison Wesley,* 1997, ISBN 0-201-63442-2.

[Korn97] G. Kornaros, C. Kozyrakis, P. Vatsolaki, M. Katevenis: “Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control”, *Proc. 17th Conf. on Advanced Research in VLSI (ARVLSI'97),* Univ. of Michigan at Ann Arbor, MI USA, Sep. 1997, pp. 127-144; http://archvlsi.ics.forth.gr/ atlasI/ atlasI_arvlsi97.ps.gz

[Kumar98] V. Kumar, T. Lakshman, D. Stiliadis: ”Beyond Best Effort: Router Architectures for the Differentiated Services of Tomorrow's Internet”, *IEEE Communications Magazine,* May 1998, pp. 152-164.

[Mavro98] I. Mavroidis: ”Heap Management in Hardware”, *Technical Report FORTH-ICS/TR-222,* Institute of Computer Science, FORTH, Crete, GR; http://archvlsi.ics.forth.gr/muqpro/heapMgt.html

[Stephens99] D. Stephens, J. Bennett, H. Zhang: ”Implementing Scheduling Algorithms in High-Speed Networks”, *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 17, no. 6, June 1999, pp. 1145-1158. .ḧttp://www.cs.cmu.edu/People/hzhang/publications.html

[Zhang95] H. Zhang: ”Service Disciplines for Guaranteed Performance in Packet Switching Networks”, *Proceedings of the IEEE,* vol. 83, no. 10, Oct. 1995, pp. 1374-1396.

[Ioan00] A. Ioannou: ”An ASIC Core for Pipelined Heap Management to Support Scheduling in High Speed Networks”, *Technical Report FORTH-ICS/TR-278,* Institute of Computer Science, FORTH, Crete, GR; http://archvlsi.ics.forth.gr