ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Δυναμική Ανακατανομή Μεγάλων Σελίδων Μνήμης για Δίκαιη Κατανομή Πόρων και Επιβολή Προτεραιοτήτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τσαπραλής Ν. Θωμάς

Επιβλέπων : Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2019

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Δυναμική Ανακατανομή Μεγάλων Σελίδων Μνήμης για Δίκαιη Κατανομή Πόρων και Επιβολή Προτεραιοτήτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Τσαπραλής Ν. Θωμάς

Επιβλέπων : Κοζύρης Νεκτάριος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Οκτωβρίου 2019 .

....	....	....
Κοζύρης Νεκτάριος	Γκούμας Γεώργιος	Πνευματικάτος Διονύσιος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2019

....

Τσαπραλής Ν. Θωμάς

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι σύγχρονοι υπολογιστές χρειάζονται περισσότερη RAM καθώς οι διεργασίες σήμερα τείνουν να έχουν μεγάλο αποτύπωμα μνήμης. Οι ολοένα και μεγαλύτερες χωρητικότητες μνήμης όμως, οδήγησαν και σε αυξημένα κόστη μετάφρασης των εικονικών διευθύνσεων. Για την αντιμετώπιση αυτού του προβλήματος, οι κατασκευαστές υλικού παρείχαν στα TLB τη δυνατότητα να υποστηρίζουν χιλιάδες εγγραφές για μεγάλα μεγέθη σελίδων (ή όπως ονομάζονται *μεγάλες σελίδες*). Οπότε η ευθύνη για την μείωση του κόστους μετάφρασης των διευθύνσεων πέρασε από το υλικό στο εκάστοτε λογισμικό.

Το Ingens είναι ένας πρόσφατος μηχανισμός που διαχειρίζεται τις μεγάλες σελίδες με τρόπο συντονισμένο και χωρίς να χρειάζεται την παρέμβαση του χρήστη. Αντιμετωπίζοντας την συνοχή της μνήμης ως πόρο που πρέπει να διανεμηθεί σε όλες τις διεργασίες και παρακολουθώντας την χρήση και την συχνότητα των σελίδων της μνήμης, το Ingens είναι σε θέση να εξαλείψει μία σειρά από παθολογίες δίκαιης συμπεριφοράς και απόδοσης, ενδημικές στην τρέχουσα υποστήριξη των μεγάλων σελίδων.

Στην τρέχουσα διπλωματική εργασία αποδεικνύουμε ότι η αδυναμία του Ingens να ανακτήσει τις μεγάλες σελίδες, μπορεί να οδηγήσει, σε ορισμένα σενάρια, σε άδικη κατανομή των μεγάλων σελίδων του συστήματος. Στην συνέχεια δείχνουμε πως αυτή η άδικη κατανομή μπορεί να οδηγήσει και σε διαφορές στην απόδοση. Λαμβάνοντας υπόψιν το γεγονός ότι σε μερικές περιπτώσεις οι αδικίες στην απόδοση είναι μη αποδεκτές (π.χ. σενάρια παροχής cloud υπηρεσιών, όπου ο πάροχος παρέχει στους πελάτους του εικονικές μηχανές του ίδιου τύπου), σχεδιάσαμε και υλοποιήσαμε έναν μηχανισμό ανακατανομής μεγάλων σελίδων, ονόματι **HPRM** (συντομογραφία της αγγλικής φράσης Huge Page Redistribution Mechanism), ο οποίος διορθώνει το ελάττωμα του Ingens. Επικυρώσαμε την λειτουργία του HPRM ποσοτικοποιώντας το βαθμό στον οποίο μπορεί να μειώσει την αδικία που υπάρχει στο σύστημα σε σύγκρισή με την προεπιλεγμένη εκδοχή του Ingens. Τέλος, συγκρίναμε την απόδοση εφαρμογών όταν αυτές έτρεχαν σε Ingens με HPRM και σε προεπιλεγμένο Ingens.

Λέξεις Κλειδιά

Εικονική μνήμη, Μεγάλες σελίδες, Διαχείριση Μνήμης, Αναδιανομή, Ingens

Abstract

Modern computing is hungry for RAM, with today's workloads that tend to have large memory footprints. Increased memory capacities have led to increased translation overheads as well. In response, hardware manufacturers provided TLBs, with thousand of entries for large page sizes (called *huge pages*). It was then the system software's responsibility to take advantage of such changes.

Ingens is a recently proposed framework that provides transparent huge page support in a principled, coordinated way. By treating memory contiguity as a resource that has to be shared across processes and by tracking utilization and access frequency of memory pages, Ingens is able to eliminate a number of fairness and performance pathologies endemic to current huge page support.

In this diploma thesis we demonstrate that Ingens' inability to reclaim huge pages could lead, in certain scenarios, to an unfair distribution of the system's available huge pages. We then show how this unfair distribution could lead to differences in performance. Motivated by the fact that in some cases (e.g. cloud provider scenarios with purchased VM instances) unfairness is unacceptable we designed and implemented a huge page redistribution mechanism, or **HPRM**, that corrects Ingens' shortcoming. We validated HPRM's functionality by quantifying the degree to which it is able to decrease the system's unfairness when compared to default Ingens'. We also evaluated how HPRM affected performance relative to default Ingens.

Keywords

Virtual memory, Huge pages, Memory management, Redistribution, Ingens

Ευχαριστίες

Με την ευκαιρία της ολοκλήρωσης της διπλωματικής μου εργασίας, θα ήθελα να ευχαριστήσω τον κ. Νεκτάριο Κοζύρη, για την ευκαιρία που μου έδωσε να εκπονήσω την διπλωματική μου εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων.Ένα μεγάλο ευχαριστώ στην Χλόη Αλβέρτη και στον Βασίλη Καρακώστα για την καθοδήγηση και υποστήριξη που μου προσέφεραν σε όλη τη διάρκεια της διπλωματικής εργασίας. Ευχαριστώ ακόμα όλους εκείνους τους φίλους, παλιούς και νέους, που έκαναν τα φοιτητικά χρόνια μια όμορφη και ξεχωριστή περίοδο.Το μεγαλύτερο ευχαριστώ είναι για την οικογένεια μου, για όλη τη στήριξη και εμπιστοσύνη που συνεχίζει να μου προσφέρει.

Θωμάς Τσαπραλής

Ιούλιος 2019

Contents

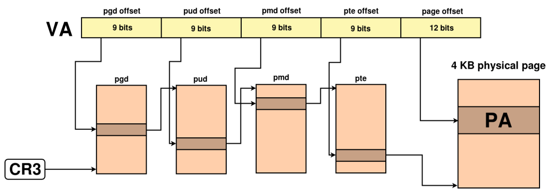
List of Figures

List of Tables

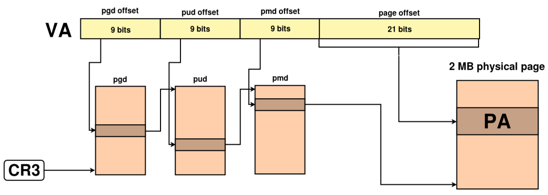
Εισαγωγη

Οι σύγχρονες εφαρμογές με μεγάλα αποτυπώματα μνήμης είναι πλέον συνήθεις [1]. Για να καλύψουν τις αυξημένες ανάγκες σε μνήμη, οι σύγχρονες πλατφόρμες υπολογιστών πλέον υποστηρίζουν μεγαλύτερες χωρητικότητες σε DRAM. Ωστόσο, οι αυξημένες χωρητικότητες προκαλούν σημαντική πρόκληση για τη μετάφραση διευθύνσεων. Όλοι οι σύγχρονοι επεξεργαστές χρησιμοποιούν πίνακες σελίδων για τις μεταφράσεις διευθύνσεων. Για να επιταχύνουν την διαδικασία της μετάφρασης οι επεξεργαστές αποθηκεύουν τις πιο πρόσφατες μεταφράσεις διευθύνσεων σε μία ειδική μνήμη cache που ονομάζεται TLB. Ωστόσο, η χωρητικότητα του TLB (αριθμός καταχωρήσεων x μέγεθος σελίδας) δεν μπορεί να κλιμακωθεί με τον ίδιο ρυθμό όπως και η DRAM, καθώς είναι πολύ δύσκολο να αυξηθεί ο αριθμός καταχωρήσεων TLB χωρίς την επιβάρυνση κόστους ή επιπρόσθετων καθυστερήσεων. Ως αποτέλεσμα, διεργασίες με μεγάλο αποτύπωμα μνήμης βιώνουν μεγάλες ποινές στην επίδοση (performance) από αστοχίες TLB και από την μετάφραση διευθύνσεων [?] όταν χρησιμοποιούν βασικές σελίδες (δηλ. 4KB). Το πρόβλημα γίνεται πιο σοβαρό με εικονικά περιβάλλοντα όπου απαιτούνται δύο επίπεδα μετάφρασης διευθύνσεων [4, 5]. Τέτοιου τύπου διεργασίες έχουν τοποθετήσει τα γενικά έξοδα μετάφρασης διευθύνσεων στο στόχαστρο των επεξεργαστών γενικής χρήσης [6, 7, 8, 9]. Σε απάντηση, οι σύγχρονες αρχιτεκτονικές έρχονται με καλύτερη υποστήριξη για μεγαλύτερα μεγέθη σελίδων, ή για *μεγάλες σελίδες* (huge pages) όπως ονομάζονται. Όπως υποδηλώνει και το όνομα, οι μεγάλες σελίδες έχουν αυξημένο μέγεθος σε σύγκριση με το μέγεθος μιας παραδοσιακής σελίδας (δηλ. 4KB). Για παράδειγμα η αρχιτεκτονική x86-64bit υποστηρίζει σελίδες 2MB και 1GB εκτός από τις σελίδες των 4KB. Στη παρούσα διπλωματική, όμως, όταν αναφερόμαστε σε μεγάλες σελίδες εννοούμε τις σελίδες μεγέθους 2MB.

Τα οφέλη που προσκομίζουν οι διεργασίες από τις μεγάλες σελίδες είναι δύο: (1) λιγότερες αστοχίες TLB, αφού μία καταχώριση TLB αντιστοιχεί σε πολύ μεγαλύτερη ποσότητα εικονικής μνήμης (π.χ. 2MB για 4KB) και (2) οι αστοχίες TLB προκαλούν μικρότερες καθυστερήσεις. Αυτό γιατί με 4KB σελίδες, η μετάφραση μιας εικονικής διεύθυνσης απαιτεί περισσότερες αναφορές στην μνήμη σε σχέση με τη περίπτωση που χρησιμοποιούνται μεγάλες σελίδες. Για παράδειγμα στην αρχιτεκτονική x86-64bit με 4KB σελίδες, η μετάφραση μιας εικονικής διεύθυνσης απαιτεί την διάσχιση μιας ιεραρχίας 4 πινάκων σελίδων (Εικόνα 0.1). Αντιθέτως με 2MB σελίδες η μετάφραση της ίδιας εικονικής διεύθυνσης απαιτεί την διάσχιση 3 πινάκων σελίδων (Εικόνα 0.2).



Εικόνα 0.1: x86-64bit: αστοχία TLB (σελίδα 4KB)



Εικόνα 0.2: x86-64bit: αστοχία TLB (σελίδα 2MB)

Παρά την εκτεταμένη υποστήριξη στο υλικό [10], οι μεγάλες σελίδες παρείχαν μη ικανοποιητικές επιδόσεις σε σημαντικές εφαρμογές [11, 12, 13, 14, 15, 16] Αυτά τα θέματα επιδόσεων οφείλονται συχνά σε ανεπαρκείς αλγόριθμους διαχείρισης των λειτουργικών συστημάτων [17, 18, 19]. Έτσι, η ευθύνη για καλύτερη υποστήριξη των μεγάλων σελίδων και βελτιωμένη απόδοση εικονικής μνήμης έχει μετατοπιστεί από το υλικό στο εκάστοτε λογισμικό των συστημάτων.

Αρχικά το Linux, υποστήριξε την χρήση των μεγάλων σελίδων μέσω μιας βιβλιοθήκης (libhugetlbfs [22]). Με αυτόν τον τρόπο το βάρος της διαχείρισης των μεγάλων σελίδων επιβάρυνε τον χρήστη, ο οποίος έπρεπε κάθε φορά να δεσμεύει εκ των προτέρων μεγάλες σελίδες του συστήματος. Δημιουργήθηκε, λοιπόν, η ανάγκη για έναν μηχανισμό διαχείρισης μεγάλων σελίδων διαφανή (*transparent*) στον χρήστη. Η ιδέα είναι ότι ο πυρήνας του λειτουργικού συστήματος εκχωρεί μεγάλες σελίδες αυτόματα, χωρίς δηλαδή την παρέμβαση του χρήστη. Η υποστήριξη ενός διαφανούς τρόπου διαχείρισης μεγάλων σελίδων [23, 24] είναι ο μοναδικός τρόπος να φέρουμε τα οφέλη των μεγάλων σελίδων σε όλες τις εφαρμογές.

Το Linux εισήγαγε έναν διαφανή μηχανισμό διαχείρισης των μεγάλων σελίδων ο οποίος ευνοεί την επίδοση των διεργασιών αλλά δεν λαμβάνει υπόψιν θέματα όπως η δίκαιη κατανομή των μεγάλων σελίδων, ο λανθάνων χρόνος του σφάλματος σελίδας (page fault latency), η εξοικονόμηση και ο κατακερματισμός της μνήμης. Σε απάντηση αυτών των ελλείψεων του Linux υλοποιήθηκε το Ingens [20]. Το Ingens είναι ένας πρόσφατος διαφανής μηχανισμός διαχείρισης μεγάλων σελίδων ο οποίος λαμβάνει υπόψιν και ισορροπεί με αποδοτικό τρόπο τα οφέλη των μεγάλων σελίδων με τα προβλήματα που αυτά δημιουργουν.

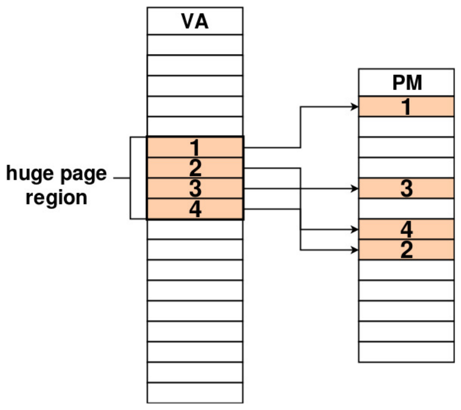
Στο πλαίσιο της παρούσας διπλωματικής εργασίας και μελετώντας προσεκτικά το Ingens ανακαλύψαμε πως το όταν το σύστημα βρίσκεται υπο πίεση μνήμης, το Ingens δεν μπορεί σε ορισμένες περιπτώσεις να πετύχει μία δίκαιη κατανομή μεγάλων σελίδων ανάμεσα στις υπό εκτέλεση διεργασίες. Έχοντας, λοιπόν, ως κίνητρο το γεγονός ότι σε ορισμένα περιβάλλοντα η άδικη κατανομή των πόρων του συστήματος είναι ανεπίτρεπτη, σχεδιάσαμε και υλοποιήσαμε έναν μηχανισμό που λύνει τον εν λόγω περιορισμό του Ingens.

Διαχείριση Μεγάλων Σελίδων

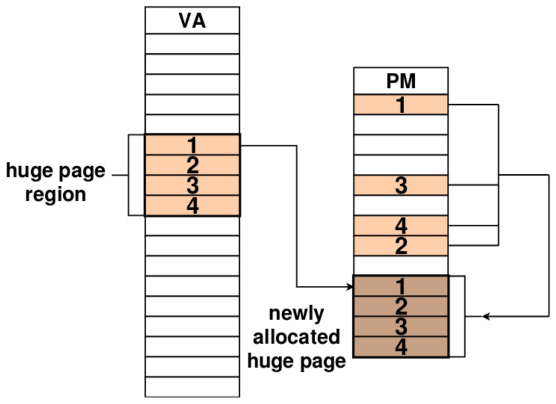
Το Linux εκχωρεί (allocates) μεγάλες σελίδες με δύο τρόπους:

1) **Σύγχρονα**, την στιγμή του σφάλματος μιας σελίδας αν ο πυρήνας μπορεί να βρεί 2MB συνεχόμενης φυσικής μνήμης.

2) **Ασύγχρονα**, με την χρήση ενός νήματος πυρήνα ονόματι *khugepaged*, σε περίπτωση που ο πυρήνας δεν είναι σε θέση να βρει τόσο ελεύθερο χώρο σύγχρονα, επειδή για παράδειγμα η μνήμη είναι κατακερματισμένη. Οπότε πυρήνας προβαίνει σε συμπίεση μνήμης (memory compaction) και αναβάλλει την εκχώρηση της μεγάλης σελίδας για αργότερα.



Εικόνα 0.3: Περιοχή μεγάλης σελίδας αντιστοιχισμένη σε διάσπαρτες σελίδες φυσικής μνήμης



Εικόνα 0.4: Το αποτέλεσμα της προαγωγής μιας περιοχής μεγάλης σελίδας σε μία μεγάλη σελίδα

Το khugepaged τρέχει περιοδικά στο παρασκήνιο σκανάρωντας τον εικονικό χώρο διευθύνσεων των διεργασιών προκειμένου να “προαγάγει” ομάδες από 512 συνεχόμενες σελίδες εικονικής μνήμης ή αλλιώς περιοχές μεγάλων σελίδων (huge page regions), σε μεγάλες σελίδες. Με τον όρο "προαγάγει" εννοούμε ότι αντιστοιχεί αυτές τις 512 συνεχόμενες σελίδες εικονικής μνήμης, οι οποίες αρχικά μπορεί να αντιστοιχούσαν σε διάσπαρτες σελίδες στην φυσική μνήμη (Εικόνα 0.3), σε 512 συνεχόμενες σελίδες φυσικής μνήμης. Για να το καταφέρει αυτό το khugepaged εκχωρεί μία μεγάλη σελίδα στην φυσική μνήμη και αντιγράφει μέσα σε αυτή τις πιθανώς διάσπαρτες σελίδες φυσικής μνήμης της περιοχής μεγάλης σελίδας (Εικόνα 0.4).

Διαχείριση Μεγάλων Σελίδων στο Linux: Περιορισμοί

Σε αυτή την ενότητα περιγράφουμε τους περιορισμούς στον διαφανή μηχανισμό διαχείρισης των μεγάλων σελίδων του Linux που οδήγησαν στην υλοποίηση του Ingens, μιας πρόσφατης λύσης στην διαχείρισης μεγάλων σελίδων για το Linux.

Λανθάνων χρόνος σφάλματος σελίδας

Όπως γνωρίζουμε από την προηγούμενη ενότητα, όταν παρουσιάζεται σφάλμα σελίδας σε μια περιοχή μνήμης, το Linux προσπαθεί να διαθέσει πρώτα μια μεγάλη σελίδα. Μόνο εάν αποτύχει η εκχώρηση μιας μεγάλης σελίδας, το Linux θα διαθέσει μια 4KB σελίδα για να ικανοποιήσει το αίτημα. Αυτή η επιθετική προσέγγιση αυξάνει την καθυστέρηση σφάλματος σελίδας για δύο λόγους:

1) Το Linux πρέπει να μηδενίσει την τεράστια σελίδα πριν την επιστρέψει στον χρήστη. Το πρόβλημα έγκειται στο γεγονός ότι οι τεράστιες σελίδες είναι 512 φορές μεγαλύτερες από τις βασικές σελίδες και, επομένως, είναι πολύ πιο αργές να μηδενιστούν.

2) Το Linux μπορεί να προχωρήσει σε σύγχρονη συμπίεση μνήμης σε μια προσπάθεια να διαθέσει μια τεράστια σελίδα τη στιγμή του σφάλματος της σελίδας. Αυτό συμβαίνει συχνά όταν η μνήμη είναι κατακερματισμένη, με αποτέλεσμα, περιοχές ελεύθερης συνεχόμενης φυσικής μνήμης δεν βρίσκονται εύκολα.

Εσωτερικός κατακερματισμός

Το Linux δεν λαμβάνει υπόψιν το αν η μεγάλη σελίδα που θα δώσει σε μία διεργασία θα χρησιμοποιηθεί στην ολότητα της με αποτέλεσμα πολλές μεγάλες σελίδες να χρησιμοποιούνται εν μέρη. Για παράδειγμα, μία διεργασία μπορεί να χρησιμοποιεί μόνο τις μισές 4K σελίδες που περιέχονται σε μία μεγάλη σελίδα. Αυτό όμως δεν σταματά το Linux από το να την εκχωρήσει στην διεργασία.

Εξωτερικός κατακερματισμός

Η άπληστη προσέγγιση κατανομής μεγάλων σελίδων του Linux καταναλώνει γρήγορα τις συνέχειες στην φυσική μνήμη αφήνοντας την υπόλοιπη φυσική μνήμη κατακερματισμένη.

Άδικη κατανομή μεγάλων σελίδων

Το νήμα πυρήνα khugepaged δεν διανέμει δίκαια τις μεγάλες σελίδες του συστήματος σε όλες τις υπό εκτέλεση διεργασίες, καθώς προάγει σε μεγάλες σελίδες όλες τις περιοχές μεγάλης σελίδας μιας διεργασίας πριν μετακινηθεί στην επόμενη. Ευνοείται, λοιπόν, η διεργασία που το khugepaged θα επιλέξει πρώτη.

Ingens: Διαχείριση Μεγάλων Σελίδων

Το Ingens είναι ένας πρόσφατος διαφανής μηχανισμός διαχείρισης μεγάλων σελίδων και βασίζεται σε δύο αρχές:

1) Η συνοχή μνήμης είναι ένας πόρος που πρέπει να παρέχεται εξίσου μεταξύ των διαδικασιών.

2) Η διαχείριση της συνέχειας μνήμης απαιτεί πληροφορίες σχετικά με τη χρήση και τη συχνότητα της πρόσβασης των σελίδων μνήμης.

Το Ingens έχει σχεδιαστεί με τρόπο που να αντιμετωπίζει τα προβλήματα που είναι ενδημικά στον τρόπο με τον οποίο το Linux διαχεριρίζεται τις μεγάλες σελίδες. Συγκεκριμένα:

1) Για να μειώσει τον λανθάνοντα του σφάλματος της σελίδας που σχετίζεται με τον σύγχρονο μηδενισμό των μεγάλων σελίδων, το Ingens απλά δεν εκχωρεί μεγάλες σελίδες σύγχρονα. Αντιθέτως όλες οι προαγωγές σε μεγάλες σελίδες πραγματοποιούνται ασύγχρονα με την χρήση ενός νήματος πυρήνα, ονόματι *promote-kth.*

2) Για να αντιμετωπίσει τον εσωτερικό κατακερματισμό το Ingens προβαίνει σε συντηρητική εκχώρηση μεγάλων σελίδων όταν η μνήμη είναι πολύ κατακερματισμένη. Συγκεκριμένα προάγει μια περιοχή μεγάλων σελίδων μόνο όταν το 90% των 4K σελίδων που περιέχονται σε αυτή έχουν αντιστοιχιστεί με σελίδες της φυσικής μνήμης. Το Ingens αντιλαμβάνεται το πότε η μνήμη είναι πολυ κατακερματισμένη μέσω μιας μετρικής (FMFI [25]).

3) Για να μετριάσει τον εξωτερικό κατακερματισμό το Ingens προβαίνει σε περιοδική προενεργή (proactive) συμπίεση μνήμης, όταν η μνήμη είναι πολύ κατακερματισμένη.

4) Για να πετύχει μία δίκαιη κατανομή των διαθέσιμων μεγάλων σελίδων του συστήματος στις υπό εκτέλεση διεργασίες, το Ingens εισάγει την έννοια της προτεραιότητας στο

promote-kth.

Το νήμα promote-kth αποτελεί την ραχοκοκκαλιά του Ingens, καθώς υλοποιεί όλες τις προαγωγές των μεγάλων σελίδων. Πρόκειται για μία διαφοροποιημένη εκδοχή του khugepaged του Linux και επιτελεί τις παρακάτω λειτουργίες:

1) Ικανοποιεί στο παρασκήνιο τα αιτήματα για μεγάλες σελίδες, με την σειρά που αυτά καταφθάνουν (*First Come First Served*), τα οποία το Linux θα ικανοποιούσε την στιγμή ενός σφάλματος σελίδας.

2) Όταν επεξεργαστεί όλα αυτά τα αιτήματα σκανάρει τον εικονικό χώρο διευθύνσεων της πιο αδικημένης διεργασίας που υπάρχει στο σύστημα για να του εκχωρήσει μεγάλες σελίδες. Επαναλαμβάνει αυτή τη διαδικασία μέχρι να επιτευχθεί μία δίκαιη κατανομή μεγάλων σελίδων.

Το Ingens εισάγει δύο μετρικές, την $Fairness\ \mathcal{M}$ που χαρακτηρίζει κάθε διεργασία και την $Unfairness\ \mathcal{O}$ που χαρακτηρίζει το σύστημα. Πιο αδικημένη ορίζεται η διεργασία p με τη μεγαλύτερη τιμή $Fairness\ \mathcal{M}$. Δίκαιη ορίζεται εκείνη η κατανομή μεγάλων σελίδων που μηδενίζει το $Unfairness\ \mathcal{O}$ του συστήματος. Σε κάθε περίπτωση το Ingens γνωρίζει ανα πάσα στιγμή κατά πόσο η υπάρχουσα κατανομή μεγάλων σελίδων είναι άδικη και σε περίπτωση που είναι, σε ποια διεργασία πρέπει να δώσει μεγάλες σελίδες για να κάνει την κατανομή δίκαιη.

Διαχείριση Μεγάλων Σελίδων στο Ingens: Περιορισμοί

Το Ingens εγγυάται δίκαιη των κατανομή μεγάλων σελίδων μεταξύ των διεργασιών. Οι συγγραφείς του Ingens το αποδεικνύουν με ένα πείραμα. Συγκεκριμένα, τρέχουν ταυτόχρονα 3 στιγμιότυπα ενός benchmark. Σε αντίθεση με το Linux, το Ingens είναι ικανό τα μοιράσει δίκαια όλες τις διαθέσιμες μεγάλες σελίδες στα 3 στιγμιότυπα με αποτέλεσμα αυτά να τελειώνουν την ίδια χρονική στιγμή.

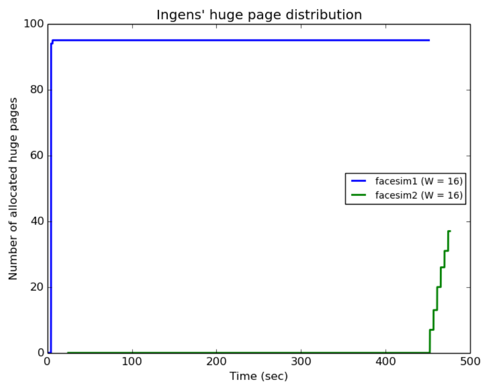
Οι συγγραφείς του Ingens υποστηρίζουν ότι η δίκαιη κατανομή των μεγάλων σελίδων είναι χρήσιμη σε σενάρια παροχής cloud υπηρεσιών, όπου ο πάροχος παρέχει στους πελάτους του εικονικές μηχανές του ίδιου τύπου. Σε αυτές τις περιπτώσεις, λοιπόν, οι μεγάλες σελίδες του συστήματος του παρόχου πρέπει να μοιραστούν ισόποσα σε όλες τις εικονικές μηχανές προκειμένου οι τελευταίες να έχουν τις ίδιες επιδόσεις. Αν δεν μοιρατούν ισόποσα, θα υπαρχει αποκλιση στις επιδόσεις των εικονικων μηχανών. Αυτό προφανως θα δυσαρεστούσε τους πελάτες οι οποίοι έχουν καθε λόγο να περιμένουν ίδιες επιδόσεις από όλες τις εικονικες μηχανές του ίδιου τύπου.

Εντούτοις, υπάρχει ένα έμφυτο πρόβλημα στο σχεδιασμό του Ingens που το αποτρέπει να πετύχει μια δικαιη κατανομη μεγαλων σελιδων ή αλλιώς μια δίκαιη κατάσταση, σε μερικές περιπτώσεις. Αυτό το σχεδιαστικό ελάττωμα πηγάζει από την ανικανότητα του Ingens να διανεμει εκ νέου τις μεγάλες σελίδες. Όταν λέμε διανέμει εκ νέου μεγάλες σελίδες εννοούμε να ξαναμοιράζει τις ήδη υπάρχουσες σελίδες (με δίκαιο τρόπο). Συγκεκριμένα οι προσπάθειες του Ingens να πετύχουν μία δίκαιη κατάσταση σταματούν όταν δεν υπάρχουν άλλες διαθέσιμες μεγάλες σελίδες στο σύστημα. Αναλυτικότερα, όταν η υπάρχουσα κατανομή των μεγάλων σελίδων είναι άδικη, το Ingens θα προσπαθήσει να τη διορθώσει δίνοντας μεγάλες σελίδες στην αδικημένη διεργασία. Ωστόσο αν το promote-kth δεν βρει επαρκή ελεύθερο χώρο στη φυσική μνήμη προκειμένου να εκχωρήσει μία μεγάλη σελίδα επειδή το σύστημα βρίσκεται υπό πίεση μνήμης, καμία μεγάλη σελίδα θα δοθεί στην αδικημένη διεργασία. Αποτελεσματικά η αδικημένη διεργασία παραμένει αδικημενη και το Ingens έχει "κολλήσει" σε αυτή την άδικη κατάσταση. Παρακάτω παρουσιάζουμε δύο σενάρια στα οποία συναντούμε το πρόβλημα αυτό.

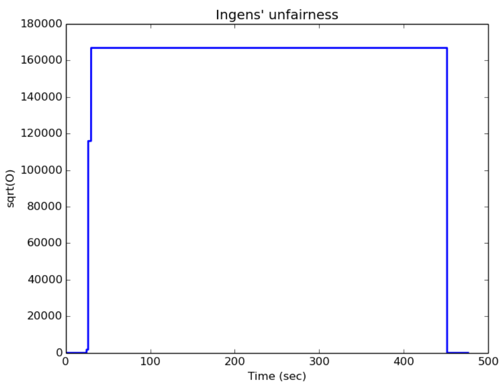
Τα πειράματα και από τα δύο σενάρια τα έχουμε τρέξει σε έναν Intel® Xeon® ProcessorE5-2630 v4 (Πίνακας 0.1). Σε κάθε σενάριο τρέχουμε 2 στιγμιότυπα του μετροπρογράμματος facesim(Parsec 3.0 benchmark [28]) με ίδια προτεραιότητα αλλά με χρονική διαφορά. Για να προσομοιώσουμε συνθήκες πίεσης μνήμης πριν από κάθε πείραμα τρέχουμε ένα ειδικά σχεδιασμένο άρθρωμα πυρήνα (kernel module). Φροντίζουμε να στρεσάρουμε την μνήμη του συστήματος τόσο ώστε οι διαθέσιμες μεγάλες σελίδες του συστήματος να φτάνουν μόνο για το ένα από τα δύο στιγμιότυπα του πειράματος.

1^o Σενάριο

Τρέχουμε το πρώτο στιγμιότυπο και όταν πάρει όλες τις διαθέσιμες σελίδες του συστήματος τρέχουμε και το δεύτερο. Παρατηρούμε ότι το δεύτερο στιγμιότυπο παίρνει μεγάλες σελίδες μόνο όταν το πρώτο τελειώσει την εκτέλεση του (Εικόνα 0.5). Το Ingens αντιλαμβάνεται πως η κατανομή των μεγάλων σελίδων είναι άδικη καθώς αυξάνεται η μετρική Unfairness (Εικόνα 0.6). Η κατανομή είναι άδικη καθώς στο σύστημα τρέχουν 2 στιγμιότυπα ίσης προτεραιότητας αλλά μόνο το ένα έχει μεγάλες σελίδες.



Εικόνα 0.5: 1^o σενάριο με προεπιλεγμένο Ingens - Διεργασίες

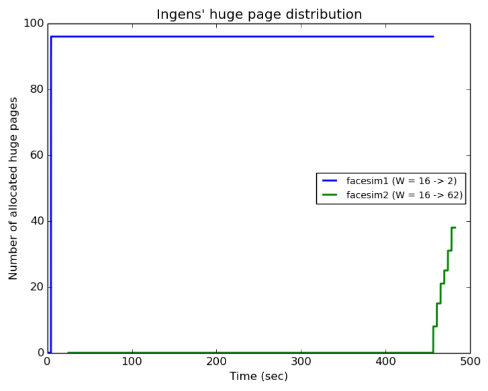


Εικόνα 0.6: 1^o σενάριο με προεπιλεγμένο Ingens - Unfairness

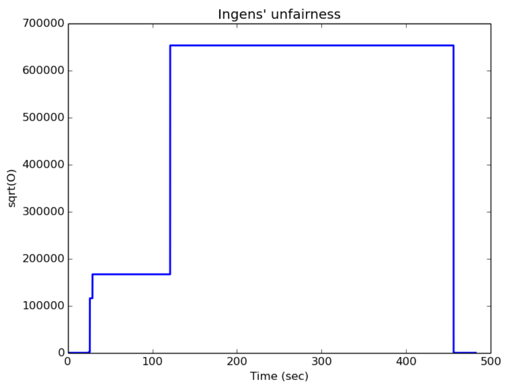
2^o Σενάριο

Επεκτείνουμε το 1ο σενάριο αλλάζοντας τις προτεραιότητες των στιγμιοτύπων δυναμικά. Συγκεκριμένα αυξάνουμε την προτεραιότητα του δεύτερου στιγμιοτύπου και μειώνουμε την προτεραιότητα του πρώτου. Το Ingens αντιλαμβάνεται την πρόσθετη αδικία από την αλλαγή της προτεραιότητας (Εικόνα 0.8) αλλά δεν μπορεί να κατανέμει ανάλογα τις μεγάλες σελίδες (Εικόνα 0.7).

Και στα δύο παραπάνω σενάρια το Ingens έχει "κολλήσει" σε μία άδικη κατάσταση. Έχοντας ως κίνητρο το γεγονός ότι σε μερικά περιβάλλοντα η άδικη κατανομή των πόρων του



Εικόνα 0.7: 2ο σενάριο με προεπιλεγμένο Ingens - Διεργασίες



Εικόνα 0.8: 2ο σενάριο με προεπιλεγμένο Ingens - Unfairness

συστήματος είναι ανεπίτρεπτη, σχεδιάσαμε και υλοποιήσαμε έναν μηχανισμό ανακατανομής μεγάλων σελίδων, ονόματι HPRM (από τον αγγλικό όρο Huge Page Redistribution Mechanism), που βοηθά το Ingens να αντιστρέψει την άδικη κατανομή των μεγάλων σελίδων στα παραπάνω σενάρια.

Huge Page Redistribution Mechanism: Σχεδίαση

Η ιδέα πίσω από το σχεδιασμό του HPRM είναι απλή. Όταν μια αδικημένη διεργασία ζητήσει μία μεγάλη σελίδα και δεν υπάρχουν διαθέσιμες μεγάλες σελίδες στο σύστημα, το HPRM αποδεσμεύει (*deallocates*) τη λιγότερο συχνά χρησιμοποιούμενη (*least frequenlty used*) μεγάλη σελίδα από την πιο ευνοημένη διαδικασία για να εξυπηρετήσει το αίτημα. Εξ ορισμού, αυτή η διαδικασία θα οδηγήσει σε μια πιο δίκαιη κατάσταση. Πρέπει επίσης να σημειωθεί ότι το HPRM αναδιανέμει μεγάλες σελίδες μόνο όταν αυτό είναι απαραίτητο, δηλαδή όταν μια αδικημένη διεργασία ζητήσει μια μεγάλη σελίδα.

Το HPRM υλοποιήθηκε μέσα στο promote-kth και επεκτείνει την λειτουργικότητα του. Ειδικότερα, το HPRM ενεργοποιείται όταν το promote-kth αδυνατεί να ικανοποιήσει ένα αίτημα μεγάλης σελίδας λόγω έλλειψης διαθέσιμων μεγάλων σελίδων. Όταν ενεργοποιηθεί, το HPRM:

1) ελέγχει εάν θα μπορούσε να επιτευχθεί δικαιότερη κατανομή μεγάλων σελίδων με την εκπλήρωση αυτού του αιτήματος μεγάλης σελίδας. Εάν όχι, το HPRM απορρίπτει το αίτημα. Εάν ναι, το HPRM

2) επιλέγει την διεργασία από την οποία θα ελευθερώσει μία μεγάλη σελίδα. Μετά,

3) βρίσκει τη λιγότερο συχνά χρησιμοποιούμενη μεγάλη σελίδα της επιλεγμένης διεργασίας. Τελικά, το HPRM

4) αποδεσμεύει αυτή την μεγάλη σελίδα από την επιλεγμένη διεργασία και την επιστρέφει στο promote-kth για να εξυπηρετήσει το αίτημα.

Huge Page Redistribution Mechanism: Αξιολόγηση

Σε αυτό το κεφάλαιο θα αξιολογήσουμε τον μηχανισμό που αναπτύξαμε τόσο σε επίπεδο επίδοσης σε σχέση με το προεπιλεγμένο Ingens όσο και σε επίπεδο δίκαιης κατανομής των μεγάλων σελίδων.

Πειραματική Πλατφόρμα

Όλα οι εκτελέσεις μετροπρογραμμάτων που παρουσιάζονται στην παρούσα διπλωματική εργασία έγιναν σε ένα NUMA node και σε επεξεργαστή Intel® Xeon® E5-2630 v4 του οποίου τα χαρακτηριστικά φαίνονται στον Πίνακα 0.1.

Οικογένεια Επεξεργαστών	Broadwell
Βασική Συχνότητα Επεξεργαστή	2.20 GHz
Αριθμός Πυρήνων	10
Αριθμός Νημάτων	20
L1 (data) Cache (ανά πυρήνα)	32KB
L2 Cache (ανά πυρήνα)	256 KB
Last Level Cache (κοινή)	25 MB

Πίνακας 0.1: Χαρακτηριστικά Intel® Xeon® E5-2630 v4.

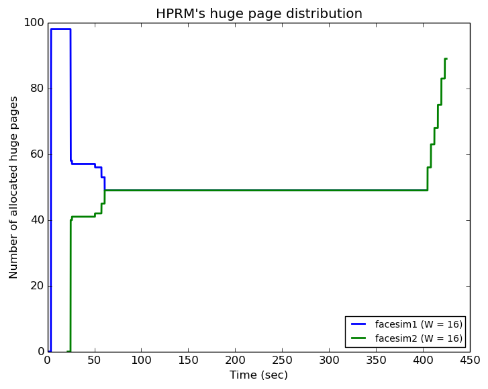
Για να προσομοιώσουμε συνθήκες πίεσης μνήμης, πριν από κάθε πείραμα τρέχουμε ένα ειδικά σχεδιασμένο άρθρωμα πυρήνα. Με αυτόν τον τρόπο περιορίζουμε τις διαθέσιμες μεγάλες σελίδες του συστήματος σε τέτοιο βαθμό ώστε να φτάνουν μόνο για το στιγμιότυπο που στα πειράματα μας τρέχουμε πρώτο.

Πως το HPRM Λύνει τους Περιορισμούς του Ingens

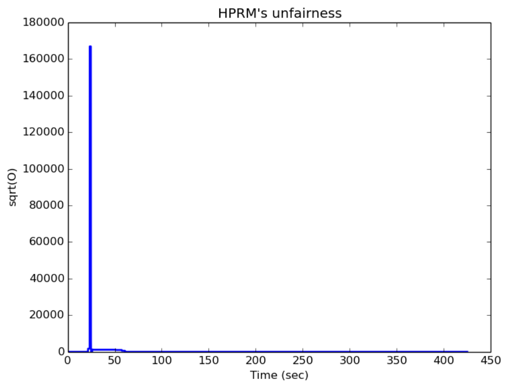
Επαναλαμβάνουμε τα σενάρια που περιγράφηκαν στο κεφάλαιο Διαχείριση Μεγάλων Σελίδων στο Ingens: Περιορισμοί για να ελέγξουμε την λειτουργικότητα του μηχανισμού μας μόνο που αυτή τη φορά τα πειράματα τα τρέχουμε όχι σε προεπιλεγμένο Ingens αλλά σε Ingens με HPRM.

1^o Σενάριο

Παρατηρούμε πως το Ingens με HPRM ανακατανέμει τις σελίδες του συστήματος όταν ξεκινά την εκτέλεση το δεύτερο στιγμιότυπο. Έτσι κάθε στιγμιότυπο παίρνει απευθείας το μερίδιο που του αναλογεί από τις διαθέσιμες μεγάλες σελίδες του συστήματος (Εικόνα 0.9). Γι'αυτό και η μετρική Unfairness μειώνεται απυεθείας (Εικόνα 0.10). Ο λόγος για τον οποίο η ανακατανομή των μεγάλων σελίδων διακόπτεται προσωρινά οφείλεται στο πόσες μεγάλες σελίδες έχει ζητήσει μέχρι εκείνη τη χρονική στιγμή το δεύτερο στιγμιότυπο.



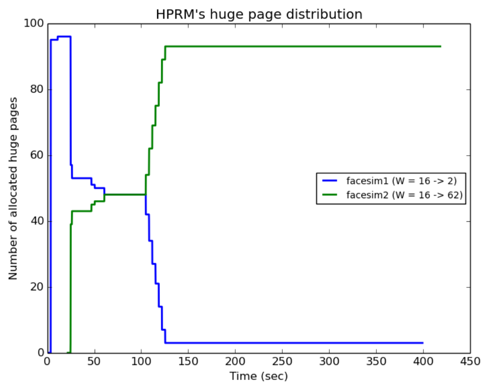
Εικόνα 0.9: 1^o σενάριο με προεπιλεγμένο Ingens - Διεργασίες



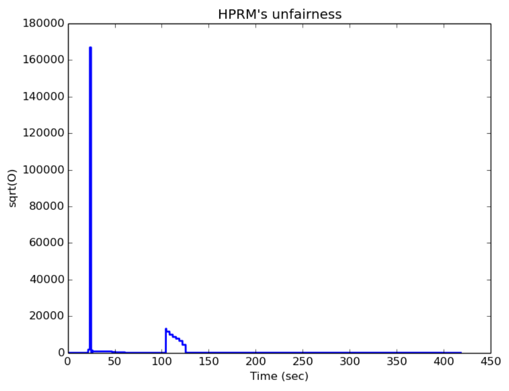
Εικόνα 0.10: 1^o σενάριο με προεπιλεγμένο Ingens - Unfairness

2^o Σενάριο

Ακόμα και όταν αλλάζουμε τις προτεραιότητες των διεργασιών δυναμικά το Ingens με HPRM έχει τη δυνατότητα να πετύχει μία δίκαιη κατάσταση μοιράζοντας τις διαθέσιμες μεγάλες σελίδες του συστήματος στα δύο στιγμιότυπα αναλογικά με τις προτεραιότητες τους. Στο στιγμιότυπο που εκτελέστηκε δεύτερο δώσαμε πολύ μεγαλύτερη προτεραιότητα σε σχέση με το πρώτο, γι'αυτό και κατέληξε να πάρει την πλειοψηφία των μεγάλων σελίδων του συστήματος (Εικόνα 0.11). Στην Εικόνα 0.12 η πρώτη αύξηση του Unfairness υποδηλώνει την εκτέλεση του δεύτερου στιγμιοτύπου και η δεύτερη την αλλαγή του βάρους. Και στις δύο περιπτώσεις το Ingens με HPRM μειώνει το Unfairness.



Εικόνα 0.11: 2^o σενάριο με προεπιλεγμένο Ingens - Διεργασίες



Εικόνα 0.12: 1^o σενάριο με προεπιλεγμένο Ingens - Unfairness

Ανάλυση Επίδοσης και Αδικίας

Μέχρι στιγμής έχουμε αποδείξει ότι το Ingens με HPRM είναι σε θέση να λύσει το περιορισμό του προεπιλεγμένου Ingens, ανακατανέμοντας τις μεγάλες σελίδες του συστήματος με έναν δίκαιο τρόπο. Σε αυτή τη παράγραφο θα εξετάσουμε αν η ανακατανομή των μεγάλων σελίδων επιφέρει και κατανομή της επίδοσης. Για να το κάνουμε αυτό ακολουθούμε την ακόλουθη διαδικασία αξιολόγησης.

Για καθένα από τα μετροπρογράμματα του Πίνακα 0.2 τρέχουμε δύο στιγμιότυπα με συγκεκριμένη χρονική μετατόπιση, σε προεπιλεγμένο Ingens και σε Ingens με HPRM. Σκοπός μας είναι να συγκρίνουμε τους χρόνους εκτέλεσης των στιγμιοτύπων στα δύο αυτά περιβάλλοντα. Πριν από κάθε πείραμα στρεσάρουμε την μνήμη εκτελώντας ένα ειδικά σχεδιασμένο άρθρωμα πυρήνα, ώστε οι διαθέσιμες μεγάλες σελίδες του συστήματος να φτάνουν μόνο για το στιγμιότυπο που τρέχει πρώτο. Ως βασικές τιμές (baseline values) θεωρούμε τους χρόνους εκτέλεσης των μετροπρογραμμάτων όταν ένα στιγμιότυπο τους τρέχει απομονωμένο σε προεπιλεγμένο Ingens (Πίνακας 0.3). Γι'αυτό και στα διαγράμματα που ακολουθούν παρουσιάζουμε μόνο επιβραδύνσεις σε σύγκριση με τις αρχικές τιμές.

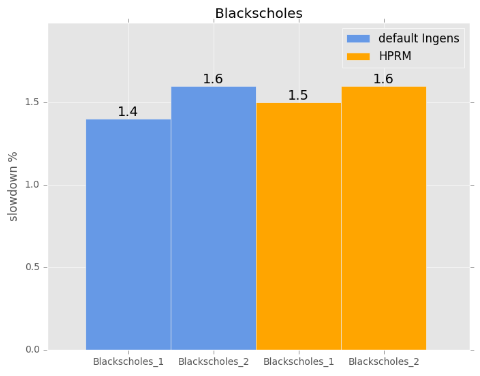
Μετροπρόγραμμα	**RSS**
Blackscholes [28]	600MB
Canneal [28]	900MB
XSBench [29]	8GB
Train [30]	30GB

Πίνακας 0.2: Μετροπρογράμματα με το RSS τους

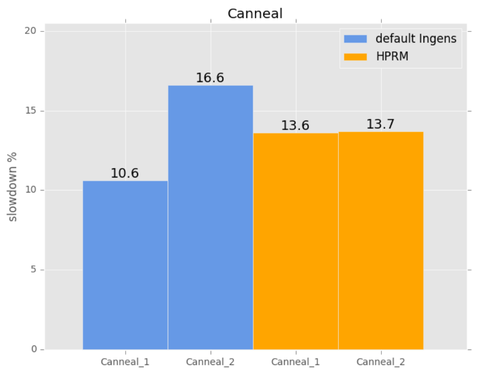
Από τις Εικόνες 0.14 , 0.15, 0.16 παρατηρούμε ότι στην περίπτωση του προεπιλεγμένου Ingens, τα στιγμιότυπα που εκτελέστηκαν πρώτα και πήραν όλες τις διαθέσιμες μεγάλες σελίδες τις οποίες διατήρησαν μέχρι και το τέλος της εκτέλεσης τους βιώνουν τις μικρότερες

	Χρόνος εκτέλεσης (δευτερόλεπτα)	
Μετροπρόγραμμα	**Προεπιλεγμένο Ingens (Baseline)**	**Linux 4K**
Blackscholes	110.7	110.8 (0.1%)
Canneal	740	798 (7.3%)
XSBench	811	958 (15.3%)
Train	854	1094 (22%)

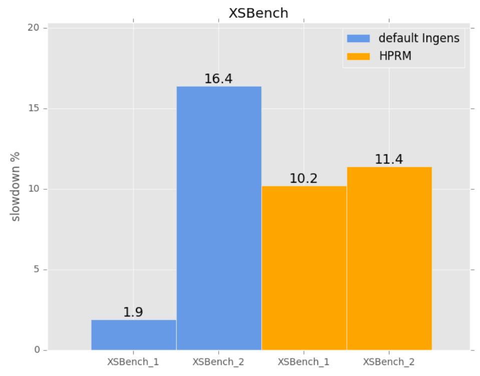
Πίνακας 0.3: Βασικές τιμές και επιβράδυνση του Linux με 4K σελίδες σε σύγκριση με το προεπιλεγμένο Ingens



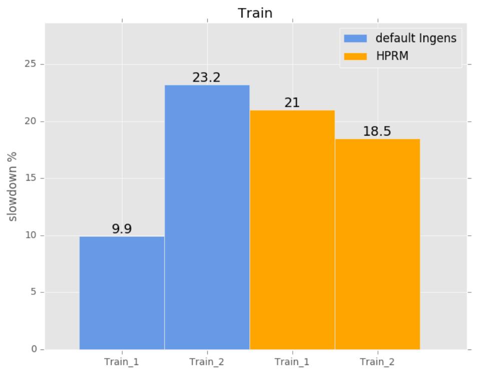
Εικόνα 0.13: Επιβραδύνσεις Blackscholes σε προεπιλεγμένο Ingens και σε Ingens με HPRM



Εικόνα 0.14: Επιβραδύνσεις Canneal σε προεπιλεγμένο Ingens και σε Ingens με HPRM.



Εικόνα 0.15: Επιβραδύνσεις XSBench σε προεπιλεγμένο Ingens και σε Ingens με HPRM



Εικόνα 0.16: Επιβραδύνσεις Train σε προεπιλεγμένο Ingens και σε Ingens με HPRM.

επιβραδύνσεις. Τα στιγμιότυπα που εκτελέστηκαν δεύτερα και παίρνουν μεγάλες σελίδες μόνο στο τέλος της εκτέλεσης τους βιώνουν τις μεγαλύτερες επιβραδύνσεις. Στην περίπτωση του Ingens με HPRM όμως παρατηρούμε ότι οι επιβραδύνσεις των στιγμιοτύπων πλησίασαν πολύ, αν και τις περισσότερες φορές η συνολική επιβράδυνση των στιγμιοτύπων αυξήθηκε. Στην Εικόνα 0.13 βλέπουμε ότι το μετροπρόγραμμα blackscholes δεν επωφελείται από τη χρήση μεγάλων σελίδων και ότι τα στιγμιότυπα σε περιβάλλον Ingens με HPRM δεν παρουσιάζουν αυξημένες επιβραδύνσεις σε σχέση με την εκτέλεση τους σε προεπιλεγμένο Ingens.

Ποσοτικοποιώντας τις διαφορές αδικίας

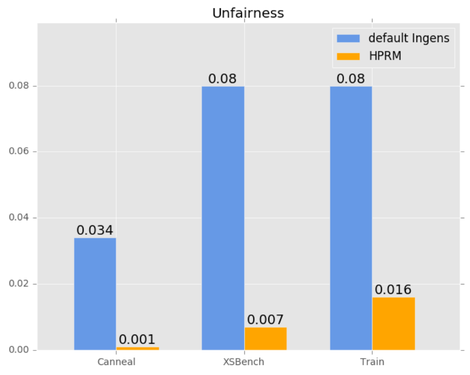
Για να ποσοτικοποιήσουμε τον βαθμό στον οποίο το Ingens με HPRM κατάφερε να μειώσει την αδικία της επίδοσης σε σχέση με το προεπιλεγμένο Ingens χρησιμοποιούμε την μετρική $Unfairness$ [31] (δεν έχει ομοιότητα με την μετρική $Unfairness\ \mathcal{O}$ του Ingens). Ορίζεται ως εξής:

$$Unfairness = \frac{\sigma_{Slowdown}}{\mu_{Slowdown}}$$

όπου $\sigma_{slowdown}$ είναι η τυπική απόκλιση των $Slowdowns$, $\mu_{slowdown}$ η μέση τιμή των $Slowdown$ και το μέγεθος $Slowdown$ ορίζεται ως εξής:

$$Slowdown = \frac{1}{Progress} = \frac{exec\_cycles_{coexecution}}{exec\_cycles_{alone}}$$

Με την μετρική αυτή ένα σύστημα είναι δίκαιο όταν όλες του οι διεργασίες βιώνουν τις ίδιες επιβραδύνσεις, οπότε η μετρική ισούται με μηδέν. Από την Εικόνα 0.17 παρατηρούμε ότι το Ingens με HPRM κατάφερε να μειώσει σε όλες τις περιπτώσεις το Unfairness που ήταν ενδημικό στο προεπιλεγμένο Ingens.



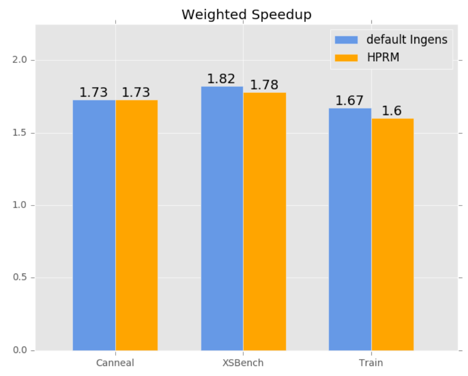
Εικόνα 0.17: Σύγκριση Unfairness μεταξύ προεπιλεγμένου Ingens και Ingens με HPRM

Ποσοτικοποιώντας τις διαφορές επίδοσης

Για να ποσοτικοποιήσουμε τον βαθμό στον οποίο το Ingens με HPRM επηρέασε την συνολική επίδοση των στιγμιοτύπων χρησιμοποιούμε την μετρική $WeightedSpeedup$ [32] που ορίζεται ως εξης:

$$Weighted\ Speedup = \sum_i \frac{IPC_i}{SingleIPC_i}$$

όπου το i ισούται με 2 καθώς στα πειράματα μας χρησιμοποιήσαμε 2 στιγμιότυπα, το IPC_i ισούται με το IPC του i στιγμιοτύπου όταν εκτελειται παράλληλα με το άλλο στιγμιότυπο και το $SingleIPC_i$ ισούται με το IPC του i στιγμιοτύπου όταν εκτελείται σε απομόνωση.



Εικόνα 0.18: Σύγκριση Weighted Speedup μεταξύ προεπιλεγμένου Ingens και Ingens με HPRM

Η μετρική αυτή υποδηλώνει μείωση στον χρόνο εκτέλεσης και ιδανικά ισούται με δύο. Από την Εικόνα 18 παρατηρούμε ότι στην γενική περίπτωση το Ingens με HPRM μειώνει το συνολική επίδοση των στιγμιοτύπων σε σχέση με το προεπιλεγμένο Ingens. Αν και οι ακριβείς λόγοι για τους οποίους παρατηρείται αυτή η συμπεριφορά έχουν αφεθεί ως μελλοντική επέκταση του μηχανισμού μπορούμε να κάνουμε την εξής επισήμανση. Όπως γνωρίζουμε το HPRM σε μία προσπάθεια να μην μειώσει την συνολική επίδοση του συστήματος αποδεσμεύει τις λιγότερο χρησιμοποιούμενες μεγάλες σελίδες της πιο ευνοημένης διεργασίας και τις επιστρέφει στο Ingens για να τις εκχωρήσει στην πιο αδικημένη διεργασία. Ωστόσο δεν γνωρίζουμε ποιες περιοχες μεγάλης σελίδας της πιο αδικημένης διεργασίας το Ingens προάγει σε μεγάλες σελίδες. Θα μπορούσε κάλλιστα να προαγάγει τις περιοχές μεγάλης σελίδας που δεν επιδέχονται αστοχίες TLB. Σε αυτήν την περίπτωση, λοιπόν, η ανακατανομή των μεγάλων σελίδων θα χειροτέρευε την συνολική επίδοση των στιγμιοτύπων.

Συμπεράσματα και Μελλοντικές Επεκτάσεις

Από τα παραπάνω πειράματα συμπεραίνουμε πως το HPRM θα μπορούσε να ενσωματωθεί στο προεπιλεγμένο Ingens, προκειμένου να συνδράμει το τελευταίο στην προσπάθεια του να πετύχει μία δίκαιη κατανομή μεγάλων σελίδων ακόμα και σε ειδικές περιπτώσεις. Αξίζει, λοιπόν, ο μηχανισμός αυτός να μελετηθεί και να εξελιχθεί περαιτέρω. Ιδιαίτερα, ως συνέχεια της παρούσας εργασίας θα ήταν ενδιαφέρον να:

- Μελετηθεί η συνέργια μεταξύ του μηχανισμού συμπίεσης μνήμης του Ingens και του HPRM καθώς και οι δύο μηχανισμοί βοηθουν στην επίτευξη μιας δίκαιης κατάστασης όταν οι μεγάλες σελίδες του συστήματος είναι περιορισμένες.
- Αξιολογηθεί η αποτελεσματικότητα του HPRM στην επιβολή προτεραιοτήτων.
- Μελετηθούν οι επιπτώσεις του Ingens σε εικονικά περιβάλλοντα.
- Μελετηθεί η χρησιμότητα του HPRM σε εικονικά περιβάλλοντα.
- Μελετηθεί το πως οι πολιτικές προαγωγής του Ingens επηρεάζουν την διαδικασία της ανακατανομής σελίδων.
- Ενσωματωθούν μετρήσεις συχνότητας πρόσβασης σελίδων στις πολιτικές προαγωγής του Ingens.

1 Introduction

Modern applications with large memory footprints are now commonplace[1]. To meet their needs modern computing platforms have increased their DRAM capacities. Nevertheless, increased capacities cause a considerable challenge for address translation. All modern processors use page tables for address translations and TLBs to cache virtual-to-physical mappings. TLB capacity (number of entries x page size), though, cannot scale at the same rate as DRAM, since it is very challenging to increase the number of TLB entries without adding extra latency and energy overheads. As a result, large memory workloads experience crippling performance penalties from TLB misses and address translation [2, 3]when they use base pages (i.e., 4KB). The problem becomes more severe with virtualized environments where two layers of address translation are needed [4, 5]. Such workloads have put address translation overheads in general-purpose processors into focus [6, 7, 8, 9]. In response, modern architectures come with better support for larger page sizes, or *huge pages*, which reduce address translation overheads by reducing the frequency of TLB misses.

Despite hardware support available[10], huge pages have provided unsatisfactory performance on important applications [11, 12, 13, 14, 15, 16]. These performance issues are often due to inadequate OS-based management algorithms [17, 18, 19]. So, the burden of better huge page support and improved virtual memory performance has shifted from the hardware to the system software.

OS-based management algorithms are called to balance the benefits huge pages offer to the running process with their shortcomings. In particular, they have to balance complex trade-offs between address translation overheads (i.e., MMU overheads), page fault latency, memory bloat and fairness. We begin by presenting two representative systems: Linux and a recent research paper Ingens [20].

Linux: Linux allocates huge pages to a process in two ways: (1) synchronously, at the time of page fault, or (2) asynchronously with the use of a dedicated kernel thread called *khugepaged* which allocates huge pages in the background. The idea is that when memory is fragmented and as a result there is not enough free contiguous space in physical memory for a huge page allocation at the time of a page fault, Linux proceeds with an optional memory compaction [21] and triggers the asynchronous way. It should be reminded that, similarly to base pages, huge pages have to be zeroed synchronously (except for COW pages) before getting mapped for security reasons.

Ingens: Ingens is a state-of-the-art memory manager for the operating system and hypervisor, that is better at handling the trade-offs associated with huge page management when compared to Linux. In summary: (1) Ingens uses an adaptive policy to balance address translation overhead and memory bloat: it uses conservative utilization-threshold based huge page allocation to prevent memory bloat under high memory pressure but relaxes the threshold to allocate huge pages aggressively under no memory pressure, to try and achieve the best of both worlds. (2) To avoid high page fault latency associated with synchronous huge page-zeroing, Ingens only allocates huge pages in the background with a dedicated kernel thread, called promote-kth which is a differentiated version of khugepaged. (3) In contrast to Linux, Ingens treats memory contiguity as a resource and employs a share-based policy to allocate huge pages fairly.

In this thesis, we demonstrate that in memory pressure scenarios, where memory compaction fails to generate enough free space for further huge page allocations, Ingens is unable to achieve a fair state when processes are spawned with a time offset or when their priorities change dynamically. To this matter, we introduce a huge page redistribution mechanism called **HPRM**. HPRM is a mechanism that triggers automatically when Ingens is "stuck" in an unfair state. When triggered, it reclaims the least frequently used huge pages, so that Ingens can distribute them in a way to reverse the unfair state. HPRM follows Ingens' line, and thus all its functionality takes place in the background. In our evaluation, we show that, in contrast to default Ingens, Ingens with HPRM is able to achieve a fair state even when huge pages are limited and processes spawn with a time offset. We also quantify how much Ingens with HPRM affected performance and unfairness when compared to default Ingens.

2 Background

2.1 The address translation process

Virtual memory is a layer of indirection between memory as seen by applications (the virtual address space) and the underlying physical memory of the hardware. Virtual memory is mapped into physical memory, i.e., memory addresses used by a process, called virtual addresses, are mapped into physical addresses in computer memory. Therefore, in order for the process to access the physical memory a virtual-to-physical address translation is interleaved. The virtual-to-physical address translations of a process are stored in a suitable data structure, called page table. Each program has its own page table and it is managed by the OS.

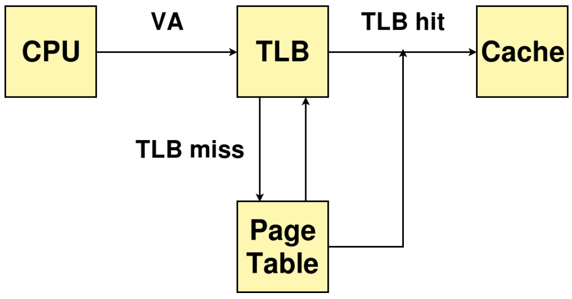


Figure 2.1: *The address translation process*

While these page tables reside in memory, the processor stores the recent virtual-to-physical address translations in a special hardware cache, called TLB (Translation Lookaside Buffer). As a result, when a virtual address needs to be translated into a physical address, the TLB is searched first. If a match is found (TLB hit), the retrieved physical address can be used to access the memory. If the requested address in not in the TLB, the translation proceeds on looking up the process' page table. This page table lookup is called a page walk. Figure 2.1 depicts the address translation process.

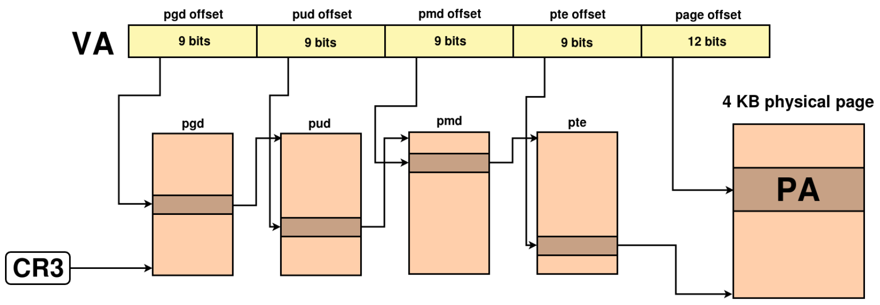


Figure 2.2: *x86_64 Page Table Layout (4K page)*

Page walks have a large performance penalty, when compared to the processor speed, as it involves reading the contents of multiple memory locations. E.g., x86-64 microarchitecture, with the use of multilevel page tables, incurs a penalty of four memory references on each TLB miss (Figure 2.2). After the physical address is determined by the page walk, the virtual to physical mapping is cached into the TLB.

2.2 Motivation for Transparent Huge Pages

Modern applications have an increasing need for memory. This need was met with the growth of DRAM's capacities. Because TLB capacities cannot scale at the same rate as DRAM, TLB misses and address translation cripples the performance of applications with large memory footprint, when these workloads use 4K pages. This problem aggravates in virtualization scenarios, where for every memory reference, two address translations are performed: guest virtual address to guest physical address and guest physical address to host physical address.

A measure to relieve big memory workloads from the address translation burden is the increase of TLB reach. TLB reach is the total memory size mapped by a TLB (number of entries x page size). An increased TLB reach would reduce the likelihood of TLB misses. TLB reach can be expanded by increasing the number of TLB entries or by increasing page size.

Increasing the number of TLB entries would be either very costly, since TLB is a hardware cache or would add extra latency. However, since TLB lookup is on the critical path of each memory access, even a small increase in latency is prohibitive. As a result modern ISAs and microarchitectures support huge pages.

As the name implies, huge pages have an increased size compare to the size of a traditional page (i.e. 4KB). For example, x86-64 supports 2MB pages and 1GB pages in addition to the default 4KB pages. In our study, though, when we refer to huge pages we mean the 2MB sized pages. Initial huge page support in Linux used a similar separate interface for huge page allocation that a developer must invoke explicitly, called hugetlbfs [22]. Nevertheless, such an approach requires manual intervention for reserving huge pages in advance and considers each application in isolation. Hence, transparent huge page (or THP for abbreviation) support is vital.

The idea is that the kernel allocates huge pages automatically, i.e., without the intervention of users and without the shortcomings of hugetlgbfs. THP support [23, 24] is the only practical way to bring the benefits of huge pages to all applications. Besides that, transparent management of huge pages best supports the multi-programmed and dynamic workloads typical of web applications and analytics where memory is contended and access patterns are often unpredictable.

2.3 Benefits of Huge Pages

There are two main reasons for which applications are running faster:

1) fewer TLB misses, since a single TLB entry maps a much larger amount of virtual memory (e.g., 2MB instead of 4KB) which leads to a larger TLB reach. The larger the TLB reach the smaller the likelihood of TLB misses.

2) TLB misses run faster. Figure 2.2 shows that in x86-64bit architecture with 4KB pages, a virtual address translation requires 4 page table traversals. Contrarily, with 2MB pages, a virtual address translation requires 3 page table traversals (Figure 2.3). As a result, page table walks run faster.

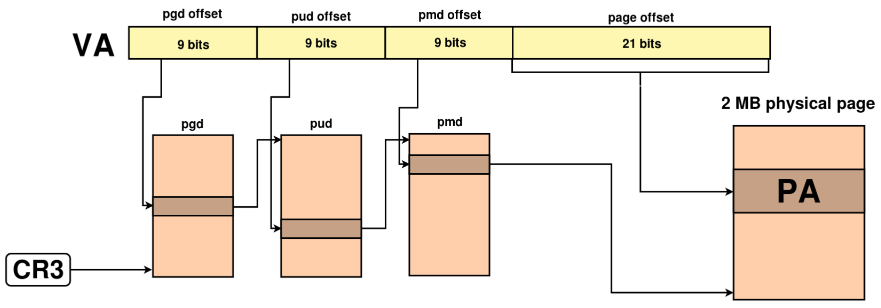


Figure 2.3: *x86_64 Page Table Layout (2MB page)*

It should be noted that the benefits of THPs are more prominent in virtualization environments, where, in contrast to native environments, address translation cost is multiple times higher.

2.4 Linux's THP management

Linux's THP allocates a huge page through two mechanisms:

1) **synchronously**, at the time of page fault, in case the kernel is able to find 2MB of contiguous physical memory, and

2) **asynchronously**, in a background kernel thread called khugepaged, in case the kernel is not able to find that much free space.

We describe those two mechanisms in detail below.

2.4.1 Synchronous Mechanism

The synchronous mechanism is implemented in the page fault critical path. When a page fault occurs, the following steps are followed:

1) The kernel tries to allocate a huge page. Depending on the success of this allocation there are two options.

2a) If the huge page allocation succeeded then the kernel maps the virtual huge page region containing the faulted page with the newly allocated huge page. For security reasons, the newly allocated huge page is zeroed synchronously (except for copy-on-write pages) before getting mapped.

2b) If the huge page allocation failed the kernel can optionally compact memory synchronously and try to allocate a huge page once again. In the meantime, the process who caused the page fault stalls. However, memory compaction can be performed asynchronously as well. In this scenario the faulted process will wake a kernel thread, *kcompactd*, to compact memory in the background so that huge pages are available in the near future. From then on, it's the responsibility of another kernel thread, khugepaged (see Asynchronous Mechanism), to use the newly freed space for huge page allocations.

3) Whatever the outcome of the huge page allocation, the kernel adds the process to a list called, *khugepaged scan list*, if it was not added already. This list is used by khugepaged (see Asynchronous Mechanism). The members of this list are processes, which are candidates for hugepage promotions.

2.4.2 Asynchronous Mechanism

The asynchronous mechanism is implemented by khugepaged. Khugepaged is a kernel thread which wakes periodically (by default every 10sec) and promotes huge pages in the background. This mechanism is useful when memory is fragmented and thus, the kernel is unable to find 2MB of contiguous physical memory at the time of the page fault.

Khugepaged's functionality is summarized in the following actions:

1) After khugepaged wakes, it selects the process located in the head of khugepage scan list.

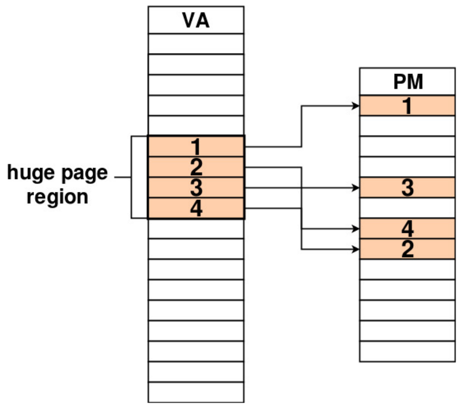


Figure 2.4: *A virtual huge page region mapped to physical memory*

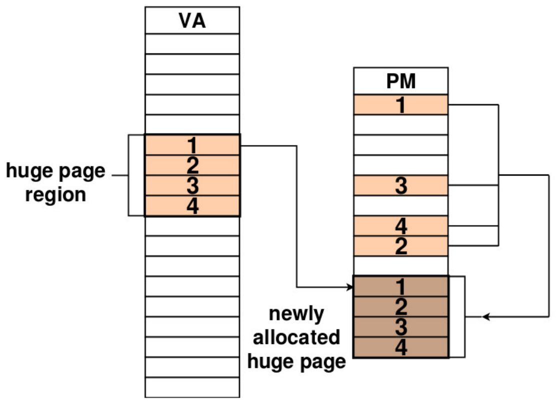


Figure 2.5: *Promotion of a huge page region to a huge page*

2) Then, khugepaged scans a predefined-sized area (by default 8 hugepage regions) of the selected process' virtual address space in order to find huge page regions that are huge page compatible, i.e., that meet certain requirements related to THP's constrains (e.g., Linux supports huge pages only for anonymous memory). Note that the physical pages backing the virtual huge page region might not be contiguous as shown in Figure 2.4.

3) If a huge page compatible region is found in the virtual address space, khugepaged tries to allocate a huge page. If the allocation is successful the kernel copies the data from the possibly discontiguous physical memory to the recently allocated huge page and maps it into the process' virtual address space (Figure 2.5). Afterwards, khugepaged frees the original discontiguous physical memory. If the allocation failed khugepaged sleeps.

4) When khugepaged finishes scanning 8 huge page regions of the selected process' virtual address space, it will sleep. The next time khugepaged wakes, it will scan the next 8 virtual huge page regions of the same process. Only after scanning the process' entire virual address space khugepaged will move to the next process of khugepage scan list.

2.5 Linux's THP management limitations

In this section we describe the limitations in Linux's THP management that led to the design Ingens, a recent huge page management solution for Linux.

Page fault latency

As we know from the previous section when a page fault occurs on an anonymous memory region, Linux always tries to allocate a huge page first. Only if that allocation fails Linux will allocate a base page to back the request. This aggressive approach increases page fault latency for two reasons:

1) Linux must zero the huge page before returning it to the user. The problem lies in the fact that huge pages are 512x larger than base pages, and thus, are much slower to clear.

2) Linux may proceed to a synchronous memory compaction in an effort to allocate a huge page at the time of the page fault. This is often the case when memory is fragmented and as a result, free contiguous physical memory in in short supply.

Memory bloat

While Linux's aggressive huge page allocation is aimed at minimizing MMU overheads, it can often lead to memory bloat. Memory bloat occurs when a process reserves more memory than it uses, e.g., it may use only half of the base pages contained in a huge page. As a result processes tend to have an increased memory footprint.

Fragmentation

The use of huge pages causes both *internal* and *external fragmentation.*

Internal fragmentation is caused when a chunk of allocated memory (e.g. a huge page) is not fully utilized and as a consequence the unusable part of it, which could be used for other memory requests, goes to waste. Internal fragmentation is the precondition for problems with memory bloat, since when a process uses a fraction of its allocated memory, it's going to have an increased memory footprint.

External fragmentation arises when free physical memory is separated in small blocks and is interleaved with allocated memory. Linux's THP management intensifies this problem because its greedy huge page allocation approach consumes quickly available physical memory contiguity leaving the remaining physical memory fragmented. The result is that free storage is available but not contiguous and thus, unusable for huge page allocations. Consequently, Linux has to start memory compaction to serve future huge page requests.

Unfair performance

Khugepaged does not distribute huge pages fairly, since it promotes all huge pages in a process before moving to the next one. This can lead to unfair performance among processes especially when huge pages become scarce ,e.g., a process may obtain all the available huge pages without leaving any for processes who run at the same time.

2.6 Ingens' THP management

Ingens is a recent huge page management framework which addresses the above problems endemic to Linux's THP management. It is based on two principles:

1) Memory contiguity is a resource that should be granted equally among processes.

2) Memory contiguity management requires information about utilization and access frequency of memory pages.

2.6.1 Ingens' Features

Ingens' features are presented below alongside with some essential implementation details. It should be noted that Ingens was built on top of Linux. Specifically, Ingens modifies the memory management code of Linux.

Fast page faults

To avoid high page fault latency associated with synchronous huge page-zeroing, Ingens, simply, does not allocate huge pages synchronously. All huge page promotions are carried out asynchronously by a dedicated kernel thread, called *promote-kth*. Promote-kth is a differentiated version of Linux's khugepaged. The most important differences between those two are:

1) Promote-kth maintains alongside khugepaged scan list two other lists called: *hugepage worklist* and *hpage scan list*. Hugepage worklist is a global list containing promotion requests from the page fault handler. In default Linux's case these requests are served synchronously. However, in Ingens' case they are buffered and served in FCFS order in the background. In addition, promote-kth has to process all the requests from hugepage worklist first, before moving to khugepage scan list. Promote-kth is not allowed to sleep for as long as there are unprocessed huge page requests in huge page worklist. Hpage scan list is another global list that contains all the processes that Ingens monitors.

2) Khugepaged always selects the process located in the head of khugepaged scan list and has to scan its whole virtual address space, 8 hugepage regions at a time, for huge

page promotions before moving to the next one. Contrariwise, promote-kth selects the process that has the highest priority in khugepaged scan list and it has to do so every time it wakes (i.e., every time it finishes scanning 8 hugepage regions of a process). This priority-based selection is performed in an effort to distribute equally huge pages among processes (see Fair Promotion).

Utilization based promotion and Frequency based demotion

To make policy decisions on promotion and demotion of huge pages , Ingens introduced two efficient mechanisms to measure the utilization of huge page regions and how frequently they are accessed. Both of them are described below:

Util bitvector (utilization tracking)

Util bitvector is a 512 bit vector that records which base pages are used within each huge-page sized memory region (an aligned huge page region is made of 512 base pages). As a result there is one util bitvector per hugepage region. Since processes generally have multiple huge page regions all the util bitvectors of a process are stored in a radix tree (there is one radix tree per process). The util bitvectors are updated by the page fault handler.

Ingens allocates huge pages for virtual huge page regions whose utilization exceeds a certain threshold (90% by default), e.g., if the number of base pages mapped in a huge page region is at least 90% of this region (i.e., at least 460 base pages mapped) then promote-kth promotes them to a huge page. This utilization threshold allows Ingens to control and therefore mitigate memory bloat. A higher utilization threshold increases address translation overheads but diminishes memory bloating. A lower utilization threshold intensifies memory bloating but decreases address translation overheads. To achieve the best of both worlds, Ingens uses an adaptive policy. It uses a conservative utilization-threshold based strategy when memory is fragmented. Nevertheless, when memory fragmentation is low Ingens behaves like Linux, promoting huge pages at the first opportunity. To quantify memory fragmentation Ingens uses the Free Memory Fragmentation Index (FMFI [25]): when FMFI<0.5 memory fragmentation is low; when FMFI>0.5 memory is highly fragmented.

Access bitvector (frequency tracking)

Access bitvector is a per-page (base or huge) 8-bit vector which records the access history of a process to its pages. In order to determine whether a page is frequently accessed or not, Ingens computes the exponential moving average (EMA [26]), defined as follows:

$$F_t = a(weight(access\ bitvector)) + (1-a)F_{t-1}$$

The weight is simply the sum of the set bits in the access bitvector, Ft is the access frequency value of a page at time t and α is a parameter between 0 and 1(Ingens sets α to 0.4). This formula suggests that the access frequency value Ft of a page at time t depends not only from the state of the access bitvector at time t but from older access frequency values as well. In any case, a high access frequency value indicates a frequently accessed page. Specifically, Ingens considers a page frequently accessed when its access frequency value Ft exceeds a certain threshold.

The access bitvectors of a process are updated by a dedicated kernel thread called *Scan-kth*, which runs periodically (by default every 5sec). To figure out if a page is currently used Scan-kth uses idle page tracking [27]. A currently used page translates to a set bit at the page's access bitvector, while a non currently used page translates to a cleared bit. To make room for the new information the bitvector shifts by one position.

Ingens uses frequency information to balance page sharing with application performance. Although huge pages increase application performance, the base pages within them can not be shared. As a consequence, a huge page has to be demoted to enable the sharing of identical base pages contained within the huge page. In contrast to KVM, which always favours memory savings over performance, Ingens denies the demotion of frequently accessed huge pages.

Proactive Compaction

With utilization based promotion Ingens mitigates internal fragmentation. To cope with external fragmentation Ingens employs proactive compaction. Proactive compaction happens in promote-kth and it is triggered when the fragmentation state of physical memory exceeds a certain threshold (FMFI>0.8 by default). With less external fragmentation Ingens is able to allocate more huge pages.

Fair Performance

To achieve fair distribution of huge pages across multiple processes, Ingens introduced a per process metric, called *memory promotion metric*. Mathematically, it is defined as follows:

$$\mathcal{M} = \frac{S}{H(f + \tau(1 - f))}$$

where S is the process' huge page share priority and it depends on the process' huge page requirement and a user defined weight W. A big value of W indicates that the corresponding process is of high priority, as far the user is concerned. Respectively, the bigger the huge page requirement of a process, the bigger its huge page share priority S gets. H is the number of bytes backed by huge pages allocated to the process. $(f + \tau(1\text{-}f))$ is a penalty factor for idle huge pages. f equals to the number of idle huge pages divided by the total number of huge pages used by this process. τ is a parameter that controls idleness penalty (i.e., the magnitude of penalty enforced to a process as a result of its idle huge pages).

Intuitively, if two processes share the same H and f, then the process with the larger S will have the larger $\mathcal{M}$ as well. A large value of $\mathcal{M}$ suggests that the corresponding process is an underprivileged one, in terms of huge page promotions, and thus its huge page requests should be prioritized. In another case, if two processes share the same S, then the process with the larger value of H(f + τ(1-f)) results in a small $\mathcal{M}$. A small value of $\mathcal{M}$ indicates that the corresponding process is an overprivileged one and therefore its huge page requests should be deprioritized.

Scan-kth updates periodically each process' memory promotion metric but promote-kth is the one who uses it. As we've seen from previous paragraph, promote-kth has to process, in FCFS order, the huge page requests from hugepage worklist first before moving to khugepage scan list. In contrast to Linux, promote-kth does not serve the huge page requests from khugepage scan list in FCFS order but rather in a memory-promotion-metric based order. That is, promote-kth selects the process that has the larger $\mathcal{M}$.

Since Ingens tries to achieve a fair distribution of huge pages among multiple processes, a metric is needed for measuring how fair the current huge page distribution is. The metric is defined as follows:

$$\mathcal{O} = \sum_{i} (\mathcal{M}_i - \overline{\mathcal{M}})^2$$

where i is the number of processes who requested huge page promotions, $\mathcal{M}_i$ is the memory promotion metric of process i and $\overline{\mathcal{M}}$ is the mean of all processes' memory promotion metric. In a perfect fair state all the $\mathcal{M}_i$ equal $\overline{\mathcal{M}}$, yielding a 0-valued $\mathcal{O}$. Promote-kth iteratively selects the process with the biggest $\mathcal{M}_i$ and scans its virtual address space to promote huge pages, in an effort to minimize $\mathcal{O}$. This iteration stops when $\mathcal{O}$ is close to zero or when there is not enough free contiguous physical memory for huge page allocations.

We should also note here that since $\mathcal{O}$ fluctuates over a vast range of values, we chose to illustrate its square root, in an effort to somewhat normalize it. Mathematically, $\mathcal{O}$ is proportional to variance and thus its square root is proportional to standard deviation, which could be used as well to evaluate fairness. Finally, from now on we shall refer to the square root of metric $\mathcal{O}$ as Ingens' *unfairness metric.*

2.6.2 Ingens' Kernel Threads

As we have seen from the previous paragraph Ingens' features is based on two kernel threads, promote-kth and scan-kth, which run periodically. We summarize their functionality below:

Promote-kth: Promote-kth's job is to fullfil promotion requests on the background. At first, it serves, in FCFS order, the hugepage requests in *hugepage worklist*. Only after promote-kth has processed all huge page requests in that list, is it allowed to sleep. If *hugepage worklist* is empty, promote-kth selects the most underprivileged process from *khugepaged scan list*, i.e., the process with the biggest promotion metric $\mathcal{M}$, and scans its virtual address space to promote huge pages. However, promote-kth does not promote huge pages blindly. It promotes only highly utilized huge pages. Promotion-metric based selection and utilization based promotion are the mechanisms that Ingens uses to achieve fair distribution of huge pages and mitigate memory bloat respectively.

Scan-kth: Scan-kth's responsibility is to update the processes' access bitvectors and promotion metrics. It does that only for the processes that are members of *hugepage scan list*. From a process' access bitvector, Ingens extracts frequency information that is used to balance page sharing with performance. The updated promotion metrics will be used by promote-kth to achieve fair distribution of huge pages.

3 Huge Page Redistribution Mechanism

3.1 Motivation

Ingens guarantees a fair distribution of huge pages among the running processes. Ingens' authors prove that with an experiment. In particular, they run 3 instances of a benchmark concurrently. In contrast to Linux, Ingens is able to share the available huge pages across the 3 instances fairly and as a result all three of them finish at the same time.

Ingens' authors point out that fair huge page distribution could be really useful in cloud provider scenarios, with purchased virtual machine instances of the same type. In such scenarios the available huge pages have to be shared fairly across all the virtual machine instances. That's because the cloud provider's customers have good reason to expect similar performance from identical virtual machine instances.

Nevertheless, there is an inherent problem with Ingens' design that forbids it from achieving a fair state, i.e., a distribution of huge pages, in certain cases. The design problem stems from Ingens' inability to redistribute huge pages. Specifically, Ingens' effort to achieve a fair distribution of the available huge pages is halted, when there aren't any available hugepages left.

As already stated, when in an unfair state, Ingens will try to reverse this situation by allocating huge pages to the most underprivileged process. However, if promote-kth is not able to find enough free contiguous physical memory for a huge page allocation due to memory pressure, even after memory compaction is performed, no huge pages will be given to the process. Memory compaction could fail to create enough space for a huge page allocation, for example, in situations with high memory utilization or fragmentation that was caused by kernel's pages which are unmovable. As a result, the process remains underprivileged and Ingens is stuck in this unfair state.

Below, we present two scenarios where this problem arises. Both scenarios' experiments were ran on a Intel® Xeon® ProcessorE5-2630 v4 (Table 4.1). In both scenarios we run 2 instances of *facesim* (Parsec 3.0 benchmark [28]) with a time offset. To simulate memory pressure we run before each experiment a kernel module that fragments a user defined portion of the system's available huge pages (for more details see section System Configuration. We adjust the portion of fragmentation, so that the system's available huge pages are enough to serve the huge page requests of only one instance.

1st Scenario

We run two instances of facesim at different times as shown in Figure 3.1. We set the same weight to both instances (W = 16), because we want them to have the same priority.

Facesim1 starts executing at t=0 sec. Since facesim1 is the only process Ingens monitors, huge pages are not contended and facesim1 ended up getting all the available huge pages very quickly. For the time that facesim1 runs alone, the system is in an ideally fair state (Ingens' unfairness metric equals to zero), simply because there aren't other processes to claim a portion of the available huge pages. Generally,in every case that Ingens monitors only one process, the system is considered to be in an ideally fair state.

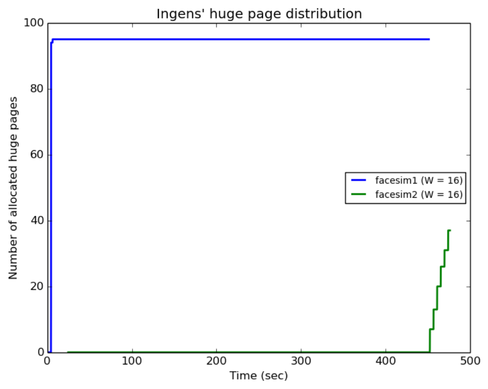


Figure 3.1: *1st Scenario in default Ingens - Processes*

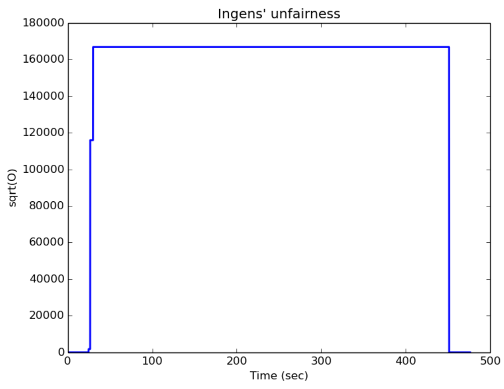


Figure 3.2: *1st Scenario in default Ingens - Unfairness*

However, when facesim2 starts execution at a later time, Ingens' unfairness metric $\mathcal{O}$ skyrocketed (Figure 3.2). That happened because Ingens recognized that there were currently 2 processes running in the system with the same priority, but only one of them had huge pages. To reverse this unfair state, Ingens tried to find additional huge pages and allocate them to facesim2, but it failed to do so, since facesim1 had already taken all of the system's available huge pages. As a result, this unfair distribution of huge pages could not be changed.

Actually, only after facesim1 finished, did Ingens' unfairness metric $\mathcal{O}$ drop to zero again and Ingens proceeded on allocating huge pages to facesim2. We could therefore imply, that this unfair state was solved, only because facesim1 finished execution. If facesim1 continued executing, this unfair state would remain as well.

2nd Scenario

We extend the first scenario by dynamically changing the priority of the running processes after spawning them. Once more, we run two instances of facesim. At first we run the first instance, facesim1, and let it get all the system's available huge pages and at a later time we run the second instance, facesim2. After facesim2 has run for a while, we also change the weight of the instances. In particular, we increase the weight of facesim2 and we decrease the weight of facesim1. We do that because, for example, facesim2 is a process with much more importance than facesim1 and we want to prioritize it.

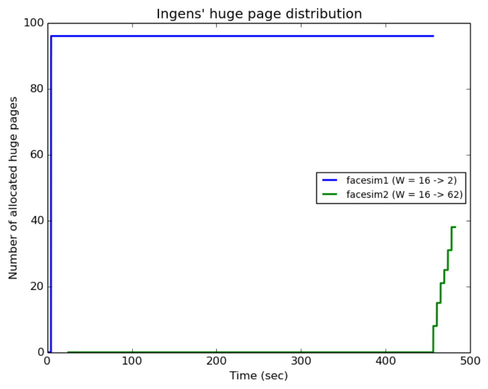


Figure 3.3: *1st Scenario in default Ingens - Processes*

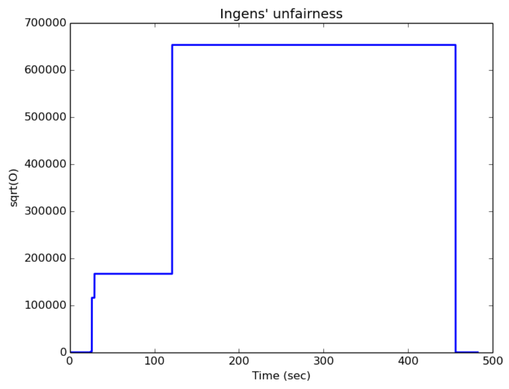


Figure 3.4: *2nd Scenario in default Ingens - Unfairness*

As Figure 3.3 depicts, Ingens is unable to adapt to priority changes as well. Even if we prioritize facesim2, it still doesn't get any huge pages. Once again, facesim1 has to finish first, before Ingens proceeds on allocating huge pages to facesim2.

The first peak in Figure 3.4 marks the time facesim2 started executing. The second peak marks the time we prioritized facesim2 and deprioritized facesim1. The second peak is much higher than the first one. The reason lies in the fact that, it is unfair for a normal priority process to not have any huge pages. But it is unfairer for a high priority process to not have any huge pages. In the beginning, Ingens' unfairness metric $\mathcal{O}$ equals to zero because only facesim1 is monitored by Ingens. At the end, Ingens unfairness metric $\mathcal{O}$ equals to zero, as well, because facesim2 is the only process Ingens monitors.

Summary

In both scenarios Ingens has stuck in an unfair state. An unfair distribution of huge pages though, translates to unfair treatment, which is not acceptable in some cases(e.g., in a cloud provider scenario) and should be treated immediately. To achieve this, we introduce a huge page redistribution mechanism (HPRM), that complements the effort Ingens' compaction makes, in order to bring the system back to a fair state.

3.2 Design and implementation

Basically the goal of HPRM is to address the unfair distribution of huge pages when there aren't any available huge pages left, even after memory compaction is performed (i.e., Ingens' compaction mechanism is unable to generate enough free physical memory contiguity for huge page allocations). It does that by redistributing the allocated huge pages across the processes in a fair way.

The idea behind HPRM's design is simple. When an underprivileged process requests a huge page and there aren't any available huge pages, HPRM deallocates the least frequently used huge page from the most privileged process to serve the request. By definition, this process will result in a fairer state. It should also be noted that HPRM's redistribution is "lazy". It addresses the unfairness when it is needed, i.e., when an underprivileged process requests a huge page.

HPRM is implemented inside promote-kth and extends its functionality. Particularly, HPRM is triggered when promote-kth is unable to fulfill a huge page request due to the lack of available huge pages. When triggered, HPRM:

1) checks whether a fairer distribution of huge pages could be achieved by fulfilling this huge page request. If no, HPRM discards the huge page request. If yes, HPRM

2) selects a process from which a huge page is going to be deallocated. It, then,

3) finds the least frequently used huge page of the selected process. Eventually, it

4) deallocates this huge page and returns it to promote-kth to serve the request.

Below we present the implementation details of each step.

Actually, the first two steps are performed simultaneously. That's because achieving a fairer state, by fulfilling a huge page request, depends from which process HPRM is going to deallocate a huge page from.

The first thing HPRM does is to check how much unfair the current distribution of huge pages is, by calculating $\mathcal{O}$. A 0-valued $\mathcal{O}$ indicates that a perfect fair state has been already achieved and therefore, there is no need for huge page redistribution. As a result, HPRM discards the huge page request and gives the control back to promote-kth. However, a greater-than-zero value of $\mathcal{O}$, shows that the current distribution of huge pages is, more or less, unfair. In this case, HPRM examines whether the fulfillment of this huge page request could lead to a smaller $\mathcal{O}$., i.e., to a fairer state. Generally, if the huge page request is made from an underprivileged process, its fulfillment will translate to to a fairer state. If, on the other hand, the huge page request is made from the most privileged process, its fulfillment will result in an unfairer state. In such cases, HPRM, once again, discards the huge page request.

Algorithm 1: HPRM's algorithm

```
Trigger: HPRM is triggered when promote-kth is unable to fulfill a huge page request
Result: If a fairer distribution can be achieved it returns to promote-kth the freed lfu huge page from the most
privileged process else it returns NULL

Calculate $\overline{\mathcal{M}}$
Calculate $\mathcal{O}$
if($\mathcal{O}$ == 0)
  return NULL
selected_process = NULL
for each process in huge page scan list:
{
  Calculate $\mathcal{M}_p$
  Calculate $\mathcal{M}_i$
  Calculate $\overline{\mathcal{M}}_{new}$
  Calculate $\mathcal{O}_{new}$
  if($\mathcal{O}_{new} < \mathcal{O}$ )
  {
    $\mathcal{O} = \mathcal{O}_{new}$
    selected_process = process
  }
}
if(!selected_process)
  return NULL
lfu_hpage = find_lfu_hpage(selected_process)
deallocate(lfu_hpage)
return lfu_hpage
```

In practice HPRM tests every process in hugepage scan list for huge page deallocation and calculates $\mathcal{O}$ in each case, so that a value of $\mathcal{O}$ corresponds to the process HPRM considers taking a huge page from. If HPRM finds a value of $\mathcal{O}$ that is less than the initial $\mathcal{O}$, then we know that a underprivileged process made the request, and thus a fairer state is achievable. However, that state, although fairer than the initial one, might not be the fairest that could have been accomplished with one huge page redistribution. To find the fairest state, HPRM proceeds on searching the rest processes in order to find the smallest $\mathcal{O}$. So, if the minimum value of $\mathcal{O}$ is less than the initial one then we know, once again, that a underprivileged process made the request and thus, HPRM will deallocate a huge page from the corresponding process to serve the request. We also know that the resulted huge page distribution is the fairest one that could have been achieved with only one huge page redistribution. If the minimum value of $\mathcal{O}$ is greater or equal than the initial one then we know that the most privileged process is the one that made the request. As a result, the huge page request is discarded.

As already stated, if the huge page request was made from an underprivileged process, HPRM selects a process and proceeds on deallocating a huge page from it. The question that arises is which huge page, in particular, HPRM will deallocate from all the huge pages the selected process may have.

In the context of our mechanism, deallocating a huge page does not imply freeing the page, i.e., returning the page to the buddy allocator, since the huge page might be in use by the process that requested it. Instead, it means migrating all the base pages contained within the huge page, to another place in physical memory. The emptied huge page can then be used by another process.

However, before migrating a base page, the kernel has to isolate it first. This isolation involves acquiring heavily contended locks. HPRM seeks to deallocate the least frequently used huge page of the selected process to avoid migrating highly utilized base pages, which would lead to additional overhead. To find the least frequently used huge page, HPRM performs a page walk. In particular HPRM, scans the selected process' pmd entries, since a single pmd entry corresponds to a huge page, and checks the frequency of every huge page using its access bitvector. Once HPRM has found the least frequently used huge page of the selected process, it deallocates the huge page and returns it to promote-kth. Then, it's promote-kth's responsibility to allocate the emptied huge page to the underprivileged process who made the request.

3.3 Example

To further explain how HPRM works and how it cooperates with default Ingens' mechanisms we provide a simple example. We assume two equally weighted processes, A and B, which run concurrently in a memory pressure environment. As a result, Ingens' promote-kth won't be able to allocate more huge pages than the existing ones. To simplify our example we also assume that process B has 2 more huge pages than process A and that neither process' huge page requests are buffered. So promote-kth has to scan itself for candidate huge page regions. The sequence of actions is as follows:

1) When promote-kth wakes it selects the most underprivileged running process, which in our case is process A, because although equally as prioritized as process B, it has two fewer huge pages. Then, promote-kth scans the virtual address space of process A for candidate huge page regions. When promote-kth finds a huge page region that meets Ingens' utilization requirements (Figure 3.5) it tries to allocate a huge page. Due to

memory pressure, that we assumed in our example, the allocation fails. Upon this failed allocation HPRM is triggered.

2) When triggered HPRM confirms that by "transferring" a huge page from process B to process A, a fairer state will be achieved. Consequently, it finds the least frequently used huge page of process B and splits it, so that the base pages contained within it can be referenced individually (Figure 3.6).

3) HPRM migrates the contained 4K pages out of the physical huge page region (Figure 3.7).

4) HPRM collapses the huge page region's unused 4K pages into one huge page (Figure 3.8). It achieved that by properly changing the base pages' metadata. It then returns that huge page to promote-kth.

5) Promote-kth moves the selected huge page region's physical pages to the huge page that HPRM provided (Figure 3.9).

6) In the end, a fair state has been achieved since both processes have the same number of huge pages (Figure 3.10).

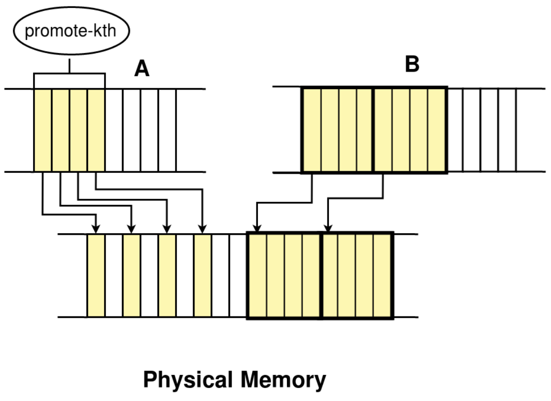


Figure 3.5: *Promote-kth finds a candidate huge page region in process A's virtual address space. HPRM is triggered*

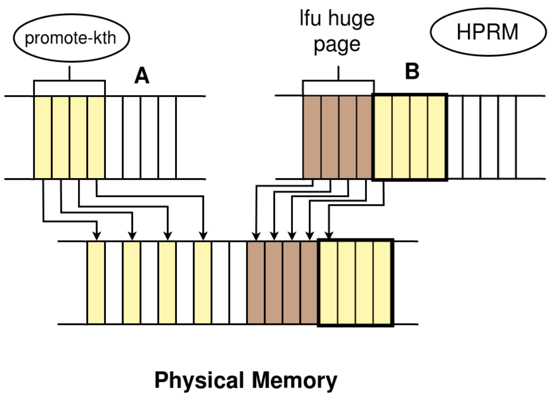


Figure 3.6: *HPRM finds and splits the lfu page of process B*

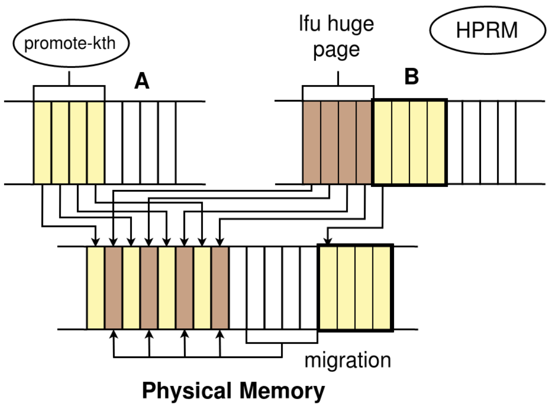


Figure 3.7: *HPRM migrates the contained 4K pages*

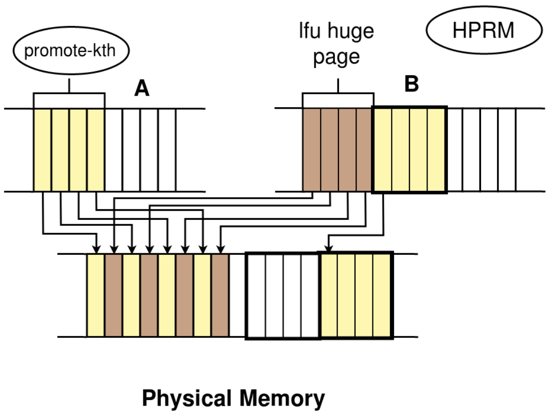


Figure 3.8: *HPRM collapses the unused 4K pages into one huge page and returns it to promote-kth*

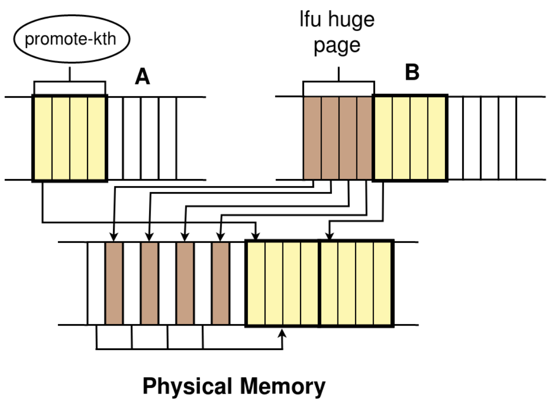


Figure 3.9: *Promote-kth allocates the unused huge page to process A.*

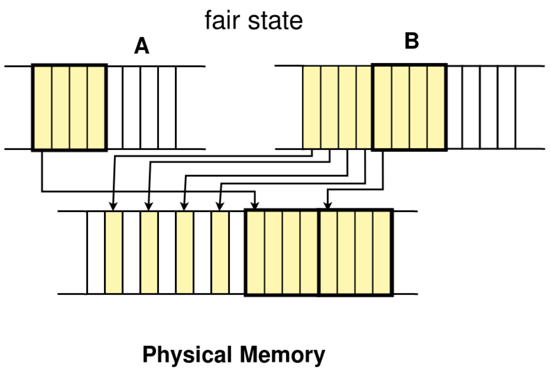


Figure 3.10: *Fair state: both processes have the same number of huge pages.*

4 Evaluation

4.1 System Configuration

All the experiments presented in this thesis were ran on a Intel® Xeon® ProcessorE5-2630 v4, within a single NUMA node. Specifications about the processor are listed in the table below.

Architecture family	Broadwell
Processor Base Frequency	2.20 GHz
Number of Cores	10
Number of Threads	20
L1 (data) Cache (per core)	32KB
L2 Cache (per core)	256 KB
Last Level Cache (shared)	25 MB

Table 4.1: *Intel® Xeon® ProcessorE5-2630 v4 specification*

In all experiments we assume that our system is under memory pressure. To simulate such conditions, before each experiment, we run a kernel module that fragments a user-defined portion of the system's available huge pages in a controlled way: it allocates 4K pages and it blocks them from reclamation, swapping and migration. As a result, Ingens' proactive compaction would not be able to generate free physical memory contiguity for further huge page allocations. Additionally, in such memory pressure conditions Ingens does not behave like Linux, which promotes huge pages aggressively, but rather adopts utilization based promotion to prevent memory bloat. Therefore, the benchmarks shown below, might not get all the huge pages that they requested but a portion of them.

In section 3.1 we presented two scenarios, where Ingens' inability to redistribute huge pages resulted in an unfair huge page distribution. With default Ingens' mechanisms this unfair state was irreversible.

Our mechanism is able to reverse the unfair state where Ingens is stuck by redistributing the already allocated huge pages in a fair way. To demonstrate that we repeat both scenarios' experiments described in section 3.1. Before each experiment, we run our micro-benchmark, so that the system's available huge pages are enough to back the huge page requests of only one instance.

4.2 How HPRM solves Ingens' limitations

In section 2.1 we presented two scenarios, where Ingens' inability to redistribute huge pages resulted in an unfair huge page distribution. With default Ingens' mechanisms this unfair state was irreversible.

Our mechanism is able to reverse the unfair state where Ingens is stuck by redistributing the already allocated huge pages in a fair way. To demonstrate that we repeat both scenarios' experiments described in section 2.1. Before each experiment, we run our micro-benchmark, so that the system's available huge pages are enough to back the huge page requests of only one instance.

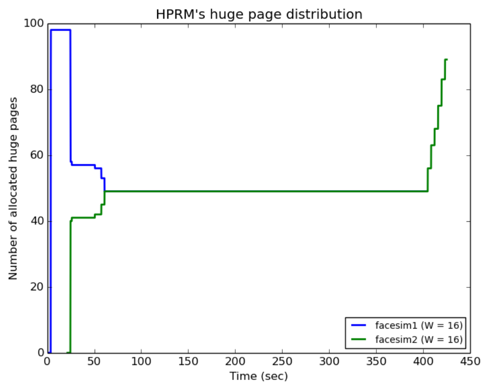
1st Scenario

We run two equally weighted instances of facesim at the times portrayed in Figure 4.1. It's not of importance the exact time facesim2 started execution. In order to show how HPRM works though, we make sure that facesim2 starts execution only after facesim1 has taken all of the system's available huge pages.

Once more, for the time that facesim1 runs alone, the system is in an ideally fair state ($\mathcal{O} = 0$). The moment facesim2 started execution, Ingens recognized that this huge page distribution is unfair and thus its unfairness metric $\mathcal{O}$ increased (Figure 4.2). From the time facesim2 started execution, all its huge page requests were buffered in hugepage worklist. It's promote-kth's responsibility to process those requests.

In default Ingens, since there aren't any available huge pages left, promote-kth would discard those requests. In our case, HPRM is able to provide huge pages to promote-kth, by deallocating them from facesim1. As a result, almost immediately, 41 huge pages from facesim1 were given to facesim2 (Figure 4.1) but then the huge page redistribution was stopped. However, from Figure 4.1, we can imply that a fair state is not achieved yet.

To understand why HPRM halted the huge page redistribution prematurely, we have to take into consideration the huge page requirement of each instance at that moment, i.e., how many huge pages they requested so far. In particular, facesim2's huge page



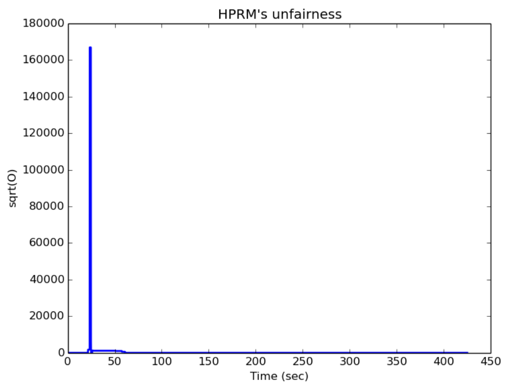


Figure 4.1: *1st Scenario in Ingens with HPRM – Processes*

Figure 4.2: *1st Scenario in Ingens with HPRM – Unfairness*

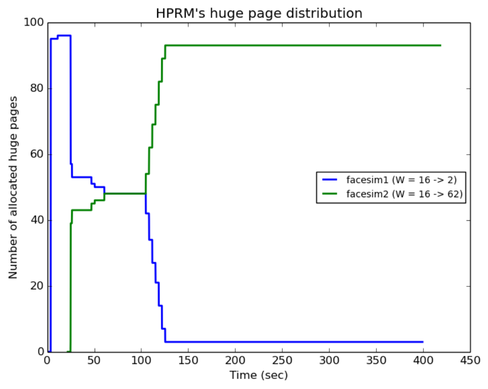
requirement was 104. So, 39.4% (41/104) of its huge page requests were served. It should be noted that in fact, facesim2's huge page requirement was 149 but scan-kth did not manage to update this value in time and thus, promote-kth uses the old value. Respectively, facesim1's huge page requirement was 149 and therefore 38.2% (57/149) of its huge page requests were served. Both percentages are so close that no further huge page redistribution would make them even closer. Hence, a fair state was achieved and Ingens unfairness metric $\mathcal{O}$ decreased to almost zero (Figure 4.2). A zeroed-value $\mathcal{O}$ would have been achieved if both instances had the same percentage of their huge page requests served, but with the current huge page requirement values this is unfeasible. In summary, HPRM stopped the huge page redistribution process, because a fair state was achieved (even though such an event can not be implied from Figure 4.1). In addition, since a fair state was achieved, there is no need for promote-kth to process additional huge page requests from facesim2. So, it deletes the remaining huge page requests of facesim2 from hugepage worklist, where they were buffered.

Facesim2's huge page requirement increased again almost instantly and thus, the huge page distribution was considered unfair, once again. This is depicted in Figure 4.2 by a small increase in $\mathcal{O}$, after the first peak. Nevertheless, huge page redistribution didn't resume immediately. The reason is that since promote-kth deleted all of facesim2's buffered huge page requests, we don't know which of facesim2's huge page regions need to be promoted to huge pages. Hence, promote-kth started scanning facesim2's virtual address space to find candidate huge page regions. Every time promote-kth found a huge page region that met Ingens' promotion requirements, HPRM catered for providing promote-kth with a huge page to back the request and so, huge page redistribution resumed. To sum up, after facesim2's huge page requirement had increased, the pauses in huge page redistribution were either due to the fact that promote-kth could not find quickly enough a huge page region that met Ingens' requirements or because promote-kth was sleeping (promote-kth scans 8 virtual huge page regions of a process and then sleeps).

In the end, a fair huge page distribution is achieved. This is illustrated in Figure 4.2 with a zeroed-value $\mathcal{O}$. It should be noted that when facesim1 finished execution, promote-kth begun to map the free huge pages to facesim2's address space, 8 huge pages at a time .

2^{nd} Scenario

We repeat the experiment shown in the above scenario with the only exception that we change the weight of the running instances dynamically, so that they have different priorities. In particular, we increased the weight of facesim2 and decreased the weight of facesim1. In that way, Ingens acknowledges facesim2 as a high priority process and facesim1 as a low priority one. It's not of importance the exact time we changed the weights of the benchmarks. However, to fully demonstrate our mechanism we make sure that before we change the weights, the system is in a fair state, i.e., both instances have a fair portion of the system's available huge pages.



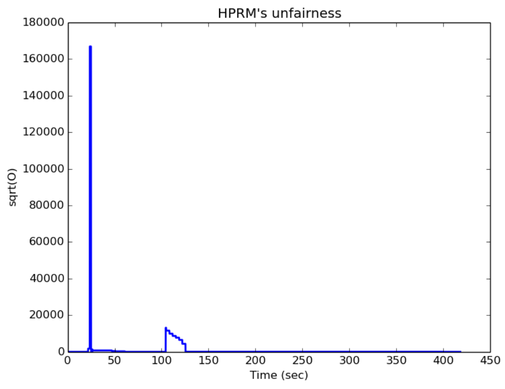


Figure 4.3: *2^{nd} Scenario in Ingens with HPRM – Processes*

Figure 4.4: *2^{nd} Scenario in Ingens with HPRM – Unfairness*

The process, by which a fair huge page distribution is achieved, is the same as it was described in the previous scenario. When facesim2 started execution its huge page requests were buffered, so that they can be processed in the background by promote-kth. To reverse the unfair distribution of huge pages promote-kth begun to serve facesim2's huge page requests with huge pages that were provided to it by HPRM. HPRM obtained those pages by deallocating the least frequently used huge pages from facesim1. Huge page redistribution was halted prematurely though, because, as far as Ingens was concerned, a fair state was achieved. Ingens' false perception was due to the fact that it was using an old value of facesim2's huge page requirement, which was lower than the original one, because scan-kth didn't manage to update this value in time. However, huge page redistribution was resumed but that time promote-kth wasn't serving facesim2's buffered huge page requests. Instead, it was searching by itself facesim2's virtual address space for candidate huge page regions. This process continued until both instances got the same portion of the system's available huge pages.

When we changed the weights of the instances, the huge page distribution was considered unfair. Promote-kth started scanning facesim2's virtual address space, 8 huge page regions at a time, for huge page allocations. The reason it selected facesim2 was because with the huge page distribution then, facesim2 was considered underprivileged and facesim1 overprivileged. In default Ingens, even if promote-kth had found a huge page region that met Ingens' promotion requirements, it wouldn't be able to proceed to a huge page allocation since there weren't any available huge pages left in the system. As a result the unfair state wouldn't be solved. In this scenario, though, HPRM is triggered. It deallocated the least frequently used huge pages of facesim1 and provided them to promote-kth. Consequently, promote-kth is able to reverse the unfair state by allocating those pages to facesim2. That process continued until a fair state was achieved, i.e., both instances got a portion of the system's available huge pages that was proportional to their new weights. Considering that facesim2's new weight (W = 62) is much larger than the one facesim1 got (W = 2), it will also acquire a much larger portion of the available huge pages (Figure 4.3).

In Figure 4.4 the first peak marks the time facesim2 started execution. Ingens' unfairness metric was increased because facesim2, although equally weighted to facesim1, didn't own any huge pages. Unfairness is decreased instantly since promote-kth served facesim2's buffered huge page requests without any pauses, until a fair state is attained. The second largest increase marks the time we changed the instances' weights. This time unfairness is decreased gradually. That is due to the fact that promote-kth allocates huge pages to facesim2 in batches of 8, since it scans facesim2's virtual address space, 8 huge page regions at a time.

4.3 How HPRM works with multiple processes

In this section we demonstrate how HPRM works with multiple processes. We extend the scenarios of section 4.2 by running an additional instance.

1st Scenario

We run three equally weighted instances of facesim. We make sure that we run the third instance only after a fair state is achieved between the other two instances, in the case our mechanism works.

Figure 4.5 illustrates how default Ingens operates under the weight of three running instances. In particular, for the time facesim1 had all of the system's available huge pages, Ingens was unable to allocate huge pages to the other two. Only after facesim1 finished execution and its huge pages were freed, did Ingens proceed on sharing those pages to the other instances, in a fair way. That is why both instances got huge pages with the same rate. When facesim2 finished execution, huge pages were allocated to facesim3 with a higher rate, since it was the only running process Ingens monitored.

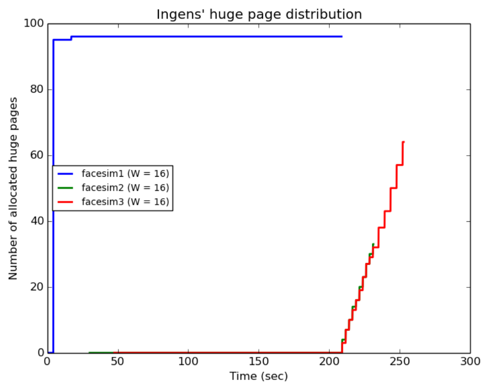


Figure 4.5: 1st *Scenario in default Ingens – Processes (3 instances)*

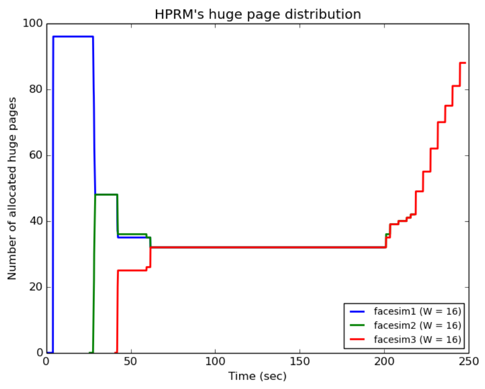


Figure 4.6: 1st *Scenario in Ingens with HPRM – Processes (3 instances)*

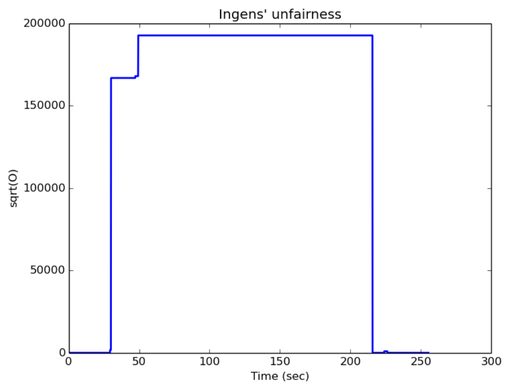


Figure 4.7: 1st *Scenario in default Ingens – Unfairness (3 instances)*

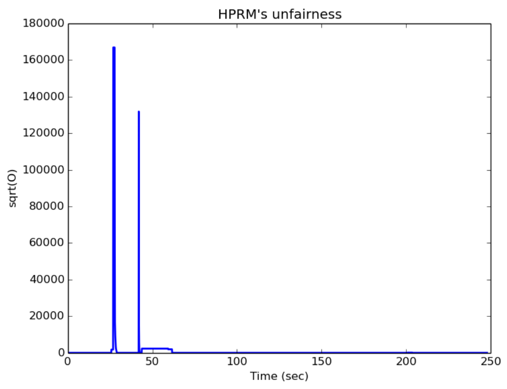


Figure 4.8: 1st *Scenario in Ingens with HPRM – Unfairness (3 instances)*

In contrast to default Ingens, HPRM is able to adapt to the new situation by redistributing the system's available huge pages again, when facesim3 started execution (Figure 4.6). It should be noted that when facesim2 started execution, huge page redistribution was carried without any pauses. That time, scan-kth was able to update facesim2's huge page requirement before a premature fair state was achieved. When facesim3 started execution, though, scan-kth didn't manage to update facesim3's huge page requirement on time and as a result huge page redistribution was stopped for a while. When facesim3's huge page requirement was updated, promote-kth started scanning its virtual address space for huge page allocations and thus, huge page redistribution was resumed. It should also be pointed out that when all instances were running concurrently, HPRM was able to share fairly the burden of huge page deallocations between facesim1 and facesim2.

In both Figure 4.7 and Figure 4.8, the first and second increase in Ingens' unfairness metric $\mathcal{O}$ denote the time facesim2 and facesim3 started execution, respectively. HPRM's first increase in $\mathcal{O}$ is equal to Ingens' first increase in $\mathcal{O}$. However, HPRM's second increase in $\mathcal{O}$ is lower than Ingens' second increase. That is due to the fact that after the first huge page redistribution HPRM performed, the system is closer to a fair state, i.e., less huge page transfers are needed to achieve a fair state.

2nd Scenario

We extend the scenario described above by changing the weight of the running instances dynamically. In the beginning all three instances were equally weighted (W = 16). We make sure that we change the weights, after a fair distribution has been attained, in the case HPRM works. We double the weight of facesim2 (W = 32) and we triple the weight of facesim3 (W = 48). We leave facesim1's weight unchanged (W = 16). In that way, we know that a fair state will be achieved when facesim2 has 2x and facesim3 3x the huge pages facesim1 owns.

Figure 4.9 illustrates Ingens' inability to adapt not only to the spawn of new instances but to weight changes as well. Only after facesim1 finished execution, was Ingens able to share its huge pages to the other 2 running instances in a fair way, i.e., in a way proportional to their new weights. Since facesim3 had a bigger weight than facesim2, Ingens allocated huge pages to facesim3 with a higher rate compared to facesim2.

Contrary to Ingens, HPRM is able to adapt to the spawning of new instances, as shown in the previous scenario, and to weight changes as well. After the weights had been changed, promote-kth started scanning facesim3's virtual address space for huge page allocations. Promote-kth selected facesim3 because with the new weights it was the most underprivileged process. As a result, huge pages were allocated to facesim3 that HPRM deallocated from facesim1, because with the new weight, it was the most privileged instance. So a fair state was achieved since every instance got a share of the system's available huge pages, proportional to its new weight (Figure 4.10). It is noteworthy that facesim1's weight didn't change but huge pages were deallocated from it. Facesim2's weight was changed, but no huge pages were allocated to or deallocated from it. Therefore the weights we set to the instances only matter in relation to each other. Once more, when facesim1 finished execution, Ingens shared its huge pages to the other two instances fairly, i.e., proportionally to their weights. When facesim2 also finished execution, all its huge pages were gradually allocated to facesim3.

Same with the previous scenario, in both Figure 4.11 and Figure 4.12 the first and second increase in Ingens' unfairness metric $\mathcal{O}$ denote the time facesim2 and facesim3 started execution, respectively. The third increase in $\mathcal{O}$ marks the time we changed the weights of the instances. The third increase in $\mathcal{O}$ is much lower in HPRM's case compared to the one in Ingens' case. That is due to the fact that after HPRM's initial redistributions the system is closer to a fairer state.

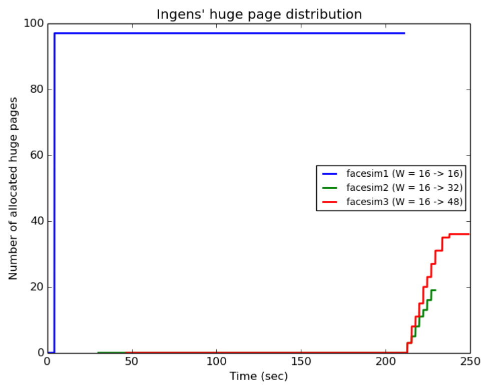


Figure 4.9: *2nd Scenario in default Ingens – Processes (3 instances)*

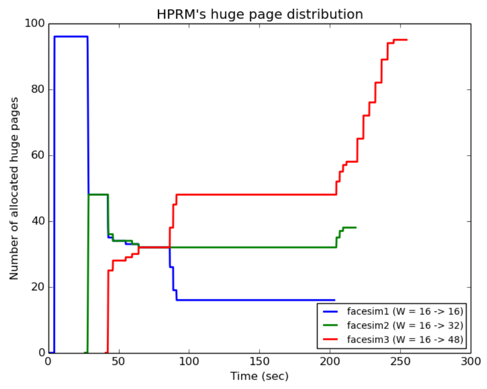


Figure 4.10: *2nd Scenario in Ingens with HPRM – Processes (3 instances)*

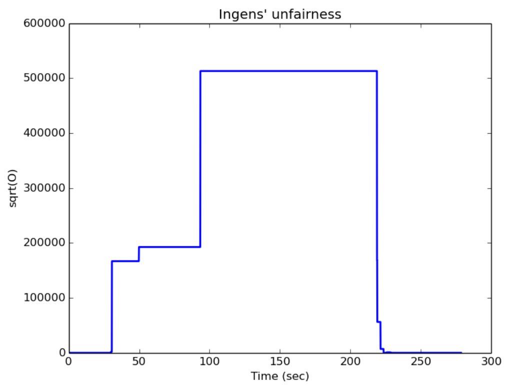


Figure 4.11: *2nd Scenario in default Ingens – Unfairness (3 instances)*

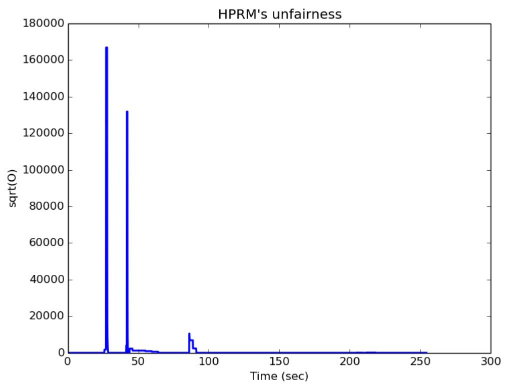


Figure 4.12: *2nd Scenario in Ingens with HPRM – Unfairness (3 instances)*

4.4 Performance and Fairness Analysis

Until now we have shown how HPRM is able to solve Ingens' fairness problem by redistributing the system's available huge pages in a fair way. Also, by minimizing Ingens' unfarness metric $\mathcal{O}$ we demonstrated that with our mechanism the resulting huge page distribution is fair.

Our initial motivation was to share the performance benefits that huge pages offer to all running processes, when Ingens could not. Therefore, we have yet to examine whether sharing the huge pages results to a sharing of performance and whether that performance sharing is indeed fair. In order to do that we use the following evaluation process.

We run 2 instances of the benchmarks presented in Table 4.2 with a specific time offset in both default Ingens and Ingens with HPRM and measure their execution times. We use time offsets because we want to make sure to run the second instance only after the first one has already taken all of the system's available huge pages. In that way, we know for sure that the only responsible for solving the unfair state is HPRM. If, on the other case, we didn't wait for the first instance to get all of the system's available huge pages, both HPRM and default Ingens' mechanisms would be co-responsible for solving the unfair state. Since we want to compare the benchmarks' execution times in default Ingens and Ingens with HPRM, we do not want to their mechanisms to be mixed.

Benchmark	**RSS**
Blackscholes [28]	600MB
Canneal [28]	900MB
XSBench [29]	8GB
Train [30]	30GB

Table 4.2: *Benchmarks with RSS*

	Execution Time (sec)	
Benchmarks	**Default Ingens (Baseline)**	**Linux 4K**
Blackscholes	110.7	110.8 (0.1%)
Canneal	740	798 (7.3%)
XSBench	811	958 (15.3%)
Train	854	1094 (22%)

Table 4.3: *Baseline values and performance slowdowns of Linux 4K when compared to default Ingens.*

To limit the number of the system's available huge pages, so that they are enough only for one instance, we run our micro-benchmark before running each set of instances. Of course, we adjust the portion of the system's huge pages we want to fragment to the RSS of the benchmark. For example, one instance of blackscholes needs about 304 huge pages and one instance of train about 14863. Therefore, in blackscholes' case we fragment the system's memory so that only 300 huge pages are available and in train's case we fragment the system's memory so that 14860 huge pages are available.

As baseline values we use the execution time of each benchmark running isolated in default Ingens (Table 4.3). Table 4.4 contains the execution times of each instance, when they are spawned with a time offset, in both default Ingens and Ingens with HPRM. The values in parenthesis represent slowdown % over baseline value. It should be pointed out that when two instances of a benchmark are running concurrently, they both experience a slowdown when compared to the case they run in isolation. That's because shared resources (in our configuration last level cache) are being contended. That's why, Table 4.4 contains only slowdowns.

Table 4.4 depicts (with the exception of blackscholes) that there are big performance differences between the instances in default Ingens' case and much smaller performance

		Execution Time (sec)	
Benchmarks	**Started execution**	**Default Ingens**	**HPRM**
Blackscholes_1	t1 = 0 sec	112.3 (1.4%)	112.4 (1.5%)
Blackscholes_2	t1 = 30 sec	112.5 (1.6%)	112.5 (1.6%)
Canneal_1	t1 = 0 sec	828 (10.6%)	856 (13.6%)
Canneal_2	t1 = 100 sec	887 (16.6%)	857 (13.7%)
XSBench_1	t1 = 0 sec	827(1.9%)	903 (10.2%)
XSBench_2	t1 = 100 sec	970 (16.4%)	915 (11.4%)
Train_1	t1 = 0 sec	948 (9.9%)	1081 (21%)
Train_2	t1 = 500 sec	1112 (23.2%)	1048 (18.5%)

Table 4.4: *Comparison between default Ingens' and Ingens with HPRM's slowdowns*

differences in HPRM's case. For example, with default Ingens, Canneal_1 has a slowdown of 10.6% and Canneal_2 has 16.6% but with HPRM Canneal_1 has 13.6% and Canneal_2 13.7% (Figure 4.14). Similar results are found with XSBench (Figure 4.15) and Train (Figure 4.16).

In default Ingens' case, the instances that were spawned first take all of system's available huge pages and maintain them until the end of their execution. The instances that were spawned at a later time get huge pages only after the first ones finish execution. As a result, there is a big performance difference between them. In contrast to default Ingens, Ingens with HPRM redistributes the system's available huge pages as soon as the second instances were spawned. As a consequence, instances experience similar performance.

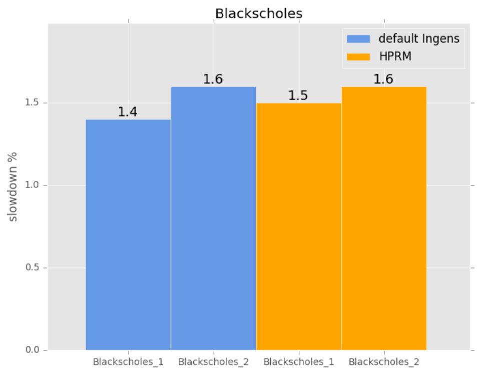


Figure 4.13: *Blackscholes' slowdowns in default Ingens and Ingens with HPRM*

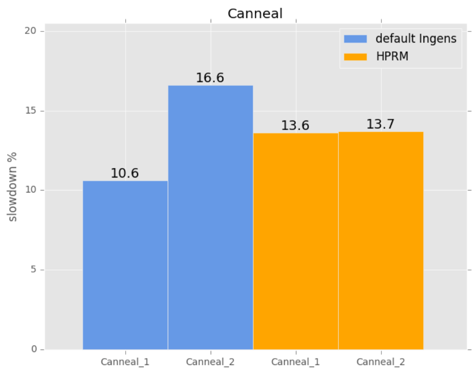


Figure 4.14: *Canneal's slowdowns in default Ingens and Ingens with HPRM*

Figure 4.13 illustrates that Blackscholes experiences no tangible performance differences whether it has huge pages or not. In default Ingens' case, even though Blackscholes_1 gets all of the system's available huge pages and maintains them until the end of its execution, it has almost exactly the same performance (performace difference = 0.2%)

as Blackscholes_2, which got huge pages only after Blackscholes_1 finished execution. No performance difference is noticed in HPRM's case, where huge pages were redistributed. So, at first glance, Figure 4.13 does not help us compare our mechanism with default Ingens. However, it demonstrates that our mechanism follows Ingens' policy, that is, functionality takes place in the background without intervening on the page fault's critical execution path. As a result, Blackscholes' instances experience no tangible slowdowns with HPRM when compared to the slowdowns of default Ingens.

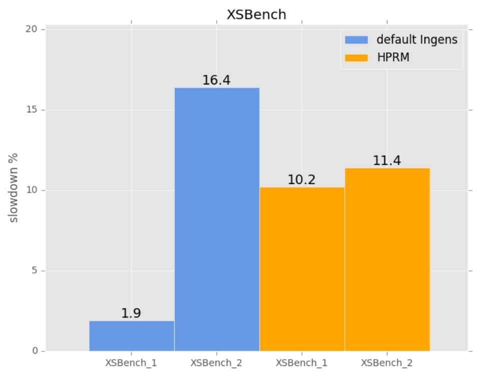


Figure 4.15: *XSBench's slowdowns in default Ingens and Ingens with HPRM*

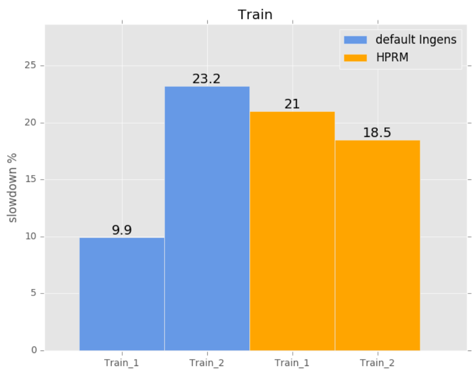


Figure 4.16: *Train's slowdowns in default Ingens and Ingens with HPRM*

In summary, HPRM was able to bring closer the performances that instances experienced in default Ingens' case. We can assume that in Canneal's case (Figure 4.14), HPRM achieved that more efficiently than in Train's case (Figure 4.16). In Canneal's case the instances' slowdowns were met in the middle, but in Train's case, the performance gap did not close that much while both of the instances' slowdowns seem rather increased. Before we rush to any conclusions though, we should first measure the degree to which HPRM affected unfairness and performance, when compared to default Ingens. We exclude Blackscholes from these measurements, since it won't provide any useful informations.

4.4.1 Quantifying unfairness

As unfairness indicator we chose the Unfairness metric [31] (which has nothing to do with the $Unfairness$ metric $\mathcal{O}$ of Ingens described in previous section) is defined as follows:

$$Unfairness = \frac{\sigma_{Slowdown}}{\mu_{Slowdown}}$$

where $\sigma_{slowdown}$ is the standard deviation of the $Slowdowns$, $\mu_{slowdown}$ is the mean

$Slowdown$ and $Slowdown$ is defined as follows:

$$Slowdown = \frac{1}{Progress} = \frac{exec\_cycles_{coexecution}}{exec\_cycles_{alone}}$$

This metric defines a system as completely fair when all tasks in the system experience the same $Slowdown$, i.e., the standard deviation σ of the $Slowdowns$ is zero. So in a ideally fair system $Unfairness$ equals to zero. We use this metric to quantify the level of unfairness in default Ingens' case and by how much HPRM was able to decrease it.

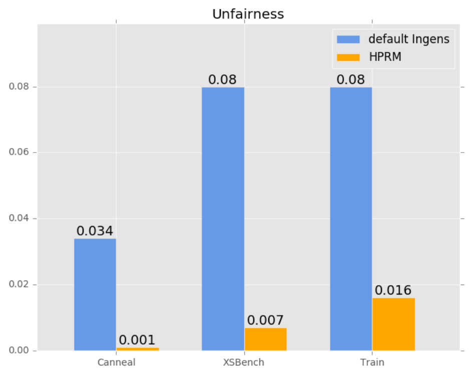


Figure 4.17: *Unfairness comparison in default Ingens and in Ingens with HPRM*

Figure 4.17 depicts that Ingens with HPRM, by redistributing the system's available huge pages, is able to reduce the system's $Unfairness$ in all occasions. In Canneal's case, $Unfairness$ dropped to almost zero. That is justified by the fact that with HPRM, both Canneal's instances experience the same performance (Figure 4.14). In Train's case $Unfairness$ decreased as well but not as much as in Canneal's case. Again, this is justified by the fact that even with HPRM, both Train's instances experience somewhat difference performance (Figure 4.16). So, in some occasions, even when HPRM redistributes the system's available huge pages in a fair way, there are still some performance differences. Studying the exact reasons there are still performance differences is left for future work. Nevertheless, we would like to make some observations.

In our experiments, the instance that was spawned first takes all the huge pages that it needs right away (promote-kth serves buffered huge page requests). However, the second instance takes only a portion of the huge pages that it needs immediately (because of HPRM). To get all the huge pages that it needs, the second instance has to wait for the first instance to finish execution and even then it gets them at a slow rate (promote-kth has to scan for candidate huge page regions). As a result, the first instance executes for a while with the maximum number of huge pages that it can get, while the second one, in most cases, does not (Figure 4.1). Hence, there is an innate difference in performance when the instances are spawned with a time offset.

4.4.2 Quantifying performance

As performance indicator we chose the $Weighted\ Speedup$ metric [32] which is defined as follows:

$$Weighted\ Speedup = \sum_i \frac{IPC_i}{SingleIPC_i}$$

where i equals to 2, since in our experiments we used 2 instances of each benchmark, IPC_i equals to the IPC of the ith application when it concurrently executes with other applications and $SingleIPC_i$ equals to the IPC of the ith application when it executes in isolation.

The $Weighted\ Speedup$ metric indicates reduction in execution time. We use this metric to quantify and compare the performance that each set of instances experience, as a whole, in both default Ingens' and HPRM's case.

Since in our experiments we have 2 instances of each benchmark running concurrently (after a specific time offset), $Weighted\ Speedup$ results from the addition of two fractions. Ideally the performance of the instance spawned first would not be affected by the second instance and therefore IPC_i would equal $SingleIPC_i$. Hence, in our experiments $Weighted\ Speedup$ has a maximum value of 2.

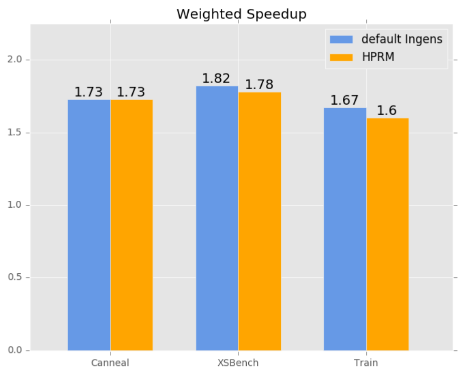


Figure 4.18: *Weighted Speedup comparison in default Ingens and in Ingens with HPRM.*

Figure 4.18 shows that HPRM lowers the cumulative performance of both instances in XSBench and Train's case. In Train's case for example, the $WeightedSpeedup$ dropped from 1.67 to 1.6. In Canneal's case, though, HPRM does not impose any performance overheads to the instances when compared to default Ingens, since the $WeightedSpeedup$ remains the same.

Recall, from the previous section, that HPRM did not cause any performance overheads to the Blackscholes' instances when compared to default Ingens (Figure 4.13) since all its functionality takes place on the background. We could assume that HPRM's performance depends from the RSS of the benchmark, since Blackscholes has a far smaller RSS when compared to Train (Table 4.2).

We could also assume that the performance overheads we see from Figure 4.18, are caused by Ingens' allocation decisions. I.e., we do not control where promote-kth will allocate the huge pages that were provided to it by HPRM. As far as we know, it could allocate them to memory regions that are not subject to TLB misses. Consequently, the corresponding instance would not experience a better performance, even though huge pages were allocated to it, and thus, huge page redistribution worsened the overall performance. In any case, studying what exactly caused these performance overheads is left for future work. The assumptions made above could be used for guidance.

5 Related Work

Huge page management is an active research area. We briefly present two recent huge page management solutions for Linux, MEGA [33] and Hawkeye [34].

MEGA

Mega's approach is to promote only huge pages that will offer a long-term performance gain. In particular, contrarily to Ingens, MEGA tracks the utilization of candidate huge page regions for longer periods. In that way, MEGA avoids the performance overheads due to the cost of frequent promotions and demotions. Additionally, MEGA's authors introduce a novel compaction algorithm that, in contrast to Ingens' compaction algorithm, seeks to move pages that are less utilized, to avoid interfering with "hot" pages, which would lead to additional overhead.

Hawkeye

Hawkeye's authors point out that Ingens has the following problems: (1) memory bloat generated in the aggressive phase of Ingens (i.e. when Ingens behaves like Linux which happens when memory fragmentation is low) remain unrecovered. (2) By allocating huge pages only in the background, Ingens nullifies an important advantage of huge pages, namely fewer page faults for access patterns exhibiting high spatial locality. (3) With Ingens two processes may have similar huge page requirements but one of them may have significantly higher TLB pressure than the other. (4) Within a process Ingens promotes huge pages through a sequential scan from lower to higher VAs.

As a response to these problems, Hawkeye: (1) tackles memory bloat by identifying and de-duplicating zero-filled baseline pages present within allocated huge pages. (2) It allocates huge pages both synchronously and asynchronously. At the same time it avoids the high page fault latency associated with synchronous huge page zeroing, by asynchronously pre-zeroing free pages. (3) In contrast to Ingens who only uses utilization informations to decide on whether it is going to promote a huge page region to a huge page, Hawkeye uses utilization, recency and frequency measurements. (4) Finally its fairness policy is based on sharing MMU overheads and not huge pages like Ingens.

Comparison with HPRM

Both MEGA and Hawkeye have not introduced a mechanism to redistribute huge pages across the running processes. Instead, in memory pressure conditions Hawkeye splits the huge pages with the most zero-filled baseline pages of the application that experiences the lowest MMU overheads and MEGA also splits the less utilized huge pages. By splitting huge pages and compacting memory both Hawkeye and MEGA could create the free space in memory for further huge page allocations that then they could be given to another process. So, in that way they can redistribute huge pages but not in a direct way like HPRM does.

6 Conclusion and Future Work

6.1 Conclusion

Ingens is a memory management redesign that addresses the problems endemic to the huge page management policy of Linux and is better at handling the trade-offs between performance, memory savings and fairness. Ingens ensures a fair huge page distribution among the running processes. In this thesis, we demonstrate that in memory pressure scenarios, where memory compaction fails to generate enough free space for further huge page allocations, Ingens is unable to achieve a fair state when processes are spawned with a time offset. To this matter we present two scenarios where Ingens' deficiency arises and causes an unfair distribution of huge pages.

Motivated by the fact that in some cases unfairness is unacceptable, (e.g. in cloud provider scenarios with purchased VM instances) we designed and implemented a huge page redistribution mechanism, called HPRM, that corrects Ingens' unfairness shortcoming. HPRM is gracefully incorporated in Ingens' existing mechanisms. We proved that in contrast to default Ingens, Ingens with HPRM is able to achieve a fair state even when multiple processes are spawned with a time offset and the system is under memory pressure.

To test whether Ingens' unfairness problem could lead to unfair performances as well, we conducted experiments using a representative set of TLB-sensitive benchmarks. For each benchmark, we spawned two instances with a time offset. To simulate memory pressure we ran a specifically developed kernel module. The experiments indicated big performance differences between the two instances. When we repeated the same experiments using Ingens with HPRM, we saw far smaller performance differences. Actually, in one case we, both instances experienced the same performance. By running an experiment in both default Ingens and Ingens with HPRM, but this time using a TLB-insensitive benchmark, we demonstrated that HPRM does not worsen the instances' execution times when compared to default Ingens. Those results were expected, since HPRM's functionality takes place in the background and does not interfere in any critical execution path.

Finally, we quantified the degree to which HPRM affected unfairness and performance when compared to default Ingens, in the experiments presented above. To achieve that, we used two metrics, one to measure unfairness and one to measure performance. Our results indicated that HPRM lowered the system's unfairness in every case. The results also showed that, in the general case, HPRM lowered the overall performance of the instances' when compared to default Ingens. Since HPRM cooperates with Ingens, further research work has to be done to find the exact cause of the performance overhead. In particular, we would find it interesting to study how Ingens' promotion decisions affect the redistribution process (see Future Work 6.2).

6.2 Future Work

Based on these results we believe that HPRM is a mechanism that complements Ingens' efforts to achieve a fair distribution of huge pages and worths to be studied and evolved. Particularly, as a continuation of the present work we would find it interesting to:

- Study the synergy between Ingens' proactive compaction and HPRM, since both mechanisms help Ingens achieve a fair state when huge pages are scarce.
- Evaluate the effectiveness of HPRM when processes' weights are changed dynamically. In this thesis we evaluated the effectiveness of HPRM to achieve a fair distribution of huge pages.
- Study the effects of Ingens' unfairness problem in virtualized environments where two layers of address translation cause additional MMU overheads. We assume that in such environments the performance gap between the instances would be larger.
- Study the usefulness of HPRM in virtualized environments.
- Study how Ingens' promotion decisions affect the redistribution process.
- Incorporate frequency measurements in Ingens' promotion decisions. As we know, HPRM deallocates the least frequently used huge pages of the most privileged process and provides them to promote-kth. Promote-kth could allocated those pages to promote the most frequently accessed huge page regions of the most underprivileged process.

Bibliography

[1] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds:A study of emerging scale-out workloads on modern hardware. In Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[2] Arkapravu Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In International Symposium on Computer Architecture (ISCA),2013.

[3] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization. In International Symposium on Microarchitecture, 2014.

[4] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII). ACM, New York, NY, USA, 26–35.
https://doi.org/10.1145/1346281.1346286

[5] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?. In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48).

[6] Mel Gorman and Patrick Healy. 2012. Performance Characteristics of Explicit Su-

perpage Support. In Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10). Springer-Verlag, Berlin, Heidelberg, 293–310.

[7] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). ACM, New York, NY, USA, 679–692.
https://doi.org/10.1145/3173162.3173203

[8] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45). IEEE Computer Society, Washington, DC, USA, 258–269.
https://doi.org/10.1109/MICRO.2012.32

[9] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17). ACM, New York, NY, USA, 469–480.
https://doi.org/10.1145/3079856.3080210

[10] 2017. Hugepages.
https://wiki.debian.org/Hugepages.

[11] 2012. Recommendation to disable huge pages for Hadoop
https://developer.amd.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[12] 2014. Recommendation to disable huge pages for NuoDB.
http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb.

[13] 2014. Why TokuDB Hates Transparent HugePages.
https://www.percona.com/blog/2014/07/23/why-tokudb-hates-transparent-hugepages/.

[14] 2019. Recommendation to disable huge pages for MongoDB.
https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[15] 2019. Recommendation to disable huge pages for Redis.
http://redis.io/topics/latency.

[16] 2019. Recommendation to disable huge pages for VoltDB.
https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[17] 2012. khugepaged eating 100% CPU.

https://bugzilla.redhat.com/show_bug.cgi?id=879801

[18] 2014. Arch Linux becomes unresponsive from khugepaged.
http://unix.stackexchange.com/questions/161858/arch-linux-becomes-unresponsive-from-khugepaged.

[19] 2015. The Black Magic Of Systematically Reducing Linux OS Jitter.
http://highscalability.com/blog/2015/4/8/the-black-magic-of-systematically-reducing-linux-os-jitter.html

[20] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, Berkeley, CA, USA, 705–721.
http://dl.acm.org/citation.cfm?id=3026877.3026931

[21] Jonathan Corbet. 2010. Memory compaction.
https://lwn.net/Articles/368869/.

[22] 2019. Libhugetlbfs: Linux man page.
https://linux.die.net/man/7/libhugetlbfs.

[23] Transparent Hugepages.
https://lwn.net/Articles/359158/. [October, 2009].

[24] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2002.

[25] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In Linux Symposium, 2005.

[26] Exponential moving average
https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average.

[27] Idle Page Tracking.
https://www.kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html

[28] PARSEC 3.0 benchmark suite.
http://parsec.cs.princeton.edu/.

[29] XSBench.
https://github.com/ANL-CESAR/XSBench

[30] Liblinear.
https://www.csie.ntu.edu.tw/ cjlin/liblinear/

[31] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit and Maria E. Gomez. Application Clustering Policies toAddress System Fairness withIntel's Cache Allocation Technology. 2017 26th International Conference on Parallel Architectures and Compilation Techniques

[32] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. https://dl.acm.org/citation.cfm?id=1194855

[33] Theodore Michailidis, Alex Delis and Mema Roussopoulos. MEGA: Overcoming Traditional Problems with OS Huge Page Management. https://dl.acm.org/citation.cfm?id=3325839

[34] Ashish Panwar, Sorav Bansal and K. Gopinath. HawkEye: Efficient Fine-grained OS Support for Huge Pages. https://dl.acm.org/citation.cfm?id=3304064