ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

SOFTWARE PREFETCHING

Μάιος 2012

Ατομική Διπλωματική Εργασία

SOFTWARE PREFETCHING

Άριστος Καραφωτιάς

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2012

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

SOFTWARE PREFETCHING

Άριστος Καραφωτιάς

Επιβλέπων Καθηγητής
Παρασκευάς Ευριπίδου

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

Μάιος 2012

Ευχαριστίες

Θα ήθελα πρώτα από όλα να εκφράσω τις ευχαριστίες μου στον επιβλέποντα καθηγητή μου, Δρ. Παρασκευά Ευριπίδου, του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου, ο οποίος με βοήθησε και με καθοδήγησε κατά την ανάπτυξη και ολοκλήρωση της διπλωματικής μου εργασίας.

Επίσης ευχαριστίες πρέπει να δοθούν στους συμφοιτητές μου οι οποίοι με βοήθησαν και μου συμπαραστάθηκαν τα τελευταία τέσσερα χρόνια, και ιδιαίτερα κατά τη διάρκεια της υλοποίησης της διπλωματικής μου εργασίας, δίνοντας μου ιδέες και λύσεις για το πώς να προχωρήσω καθώς και το πώς να αντιμετωπίσω τα διάφορα προβλήματα που κατά καιρούς εμφανίζονταν.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για τη στήριξη
που μου έδιναν καθ' όλη τη διάρκεια αυτής της χρονιάς.

Περίληψη

Η απόδοση των επεξεργαστών έχει μεγαλώσει με πολύ πιο γρήγορο ρυθμό από την απόδοση της μνήμης τις τελευταίες τρεις δεκαετίες. Αυτό προκάλεσε ένα μεγάλο χάσμα μεταξύ τους και έκανε την μνήμη να αποτελεί σημαντικό bottleneck της απόδοσης. Το πρόβλημα αυτό χειροτερεύει με την ταχεία ανάπτυξη των πολυπύρηνων επεξεργαστών, διότι στους πολλαπλούς πυρήνες η μνήμη διαμοιράζεται μεταξύ τους . Παραδοσιακά οι μνήμες cache χρησιμοποιούνταν για την βελτίωση της απόδοσης των προσβάσεων της μνήμης χρησιμοποιώντας την αρχή της τοπικότητας. Εν τούτης, οι βελτιστοποιήσεις είναι απαραίτητες για την βελτίωση της χρήσης των μνήμων cache και για την ελαχιστοποίηση των cache misses.

Γι αυτό και στο πλαίσιο της Ατομικής Διπλωματικής μου εργασίας θα ασχοληθώ με το prefetching των δεδομένων που θεωρείται ένας αποτελεσματικός τρόπος να για να μειώσει την καθυστέρηση στην πρόσβαση των δεδομένων που προκαλείται από τα cache misses και να γεφυρώσει το χάσμα απόδοσης μεταξύ του επεξεργαστή και της μνήμης.

Αυτή η έρευνα πάνω στο prefetching γίνεται και με σκοπό να μπορεί να ενσωματωθεί αργότερα μια παρόμοια υλοποίηση πάνω στο Thread Scheduling Unit του Data-Driven Multithreading μοντέλου που αποτελεί ένα ερευνητικό Project του πανεπιστημίου μου. Δηλαδή θα πρέπει να μπορεί να εφαρμοστεί το Software Prefetching πάνω στα Dthreads αυτού του Data-Driven Multithreading μοντέλου.

Αξιολογώντας τα αποτελέσματα μας βλέπουμε ότι το Software Prefetching αποδείχθηκε ότι εάν εφαρμοστεί σωστά και κάτω από τις κατάλληλες προϋποθέσεις μπορεί να φέρει θετικά αποτελέσματα στην εκτέλεση κάποιου προγράμματος μας. Ωστόσο πρέπει να λάβουμε υπόψη μας κάποιους σημαντικούς παράγοντες που θα επηρεάσουν σημαντικά την χρήση του Software Prefetching.

Περιεχόμενα

Κεφάλαιο **1** **Εισαγωγή**……. **9**

1.1 Γενική εισαγωγή στο Prefetching 9

1.2 Το μοντέλο DDM 10

1.2.1 Το Thread Synchronization Unit 12

1.2.2 Το Codeblock 13

1.2.3 Cacheflow 14

Κεφάλαιο **2** **Prefetching**……. **17**

2.1 Θέματα Prefetching 17

2.1.1 Τί να κάνουμε Prefetch 17

2.1.2 Πότε να κάνουμε Prefetch 18

2.1.3 Πού να κάνουμε Prefetch 18

2.2 Hardware και Software Prefetching 19

Κεφάλαιο **3** **Cache memory……..**……. **20**

3.1 Η μνήμη Cache 20

3.2 Cache Associativity 21

3.2.1 Direct Mapped 21

3.2.2 2-Way Set Associative 22

3.2.3 4-Way Set Associative 22

3.2.4 Fully Associative 22

3.3 Cache Entries 23

3.4 Απόδοσης της μνήμης Cache 23

3.5 Cache Misses 24

3.5.1 Compulsory misses 24

3.5.2 Capacity misses 24

3.5.3 Conflict misses 25

3.6 Cache pollution και eviction 25

3.7 Inclusive Cache και Exclusive cache 26

Κεφάλαιο **4** **Τρόποι-Συναρτήσεις για Software Prefetching……...**…… **27**

4.1 void __builtin_prefetch 27

4.2 void _mm_prefetch 28

4.3 GCC inline assembly 30

Κεφάλαιο **5** **Υλοποίηση** ……...…… **32**

5.1Τρόπος υλοποίησης 32

5.2 Παράδειγμα Prefetching 33

5.3 Σύστημα μετρήσεων 33

5.4 Τρόπος συλλογής μετρήσεων-αποτελεσμάτων 34

ΚΕΦΑΛΑΙΟ **6** **Μετρήσεις-Αποτελέσματα..**…….. **35**

6.1 Μετρήσεις κάθε πρόγραμμα 35

6.1.1 Απλό παράδειγμα αθροίσματος στοιχείων πίνακα. 35

6.1.2 Πρόσθεση πινάκων 36

6.1.3 Πολλαπλασιασμός πινάκων 36

6.1.4 Μετάθεση πινάκων 36

6.1.5 Παράλληλο πρόγραμμα αθροίσματος στοιχείων πίνακα. 37

6.2 Γραφικά Αποτελέσματα 37

6.2.1 Απλό παράδειγμα αθροίσματος στοιχείων πίνακα. 37

6.2.2 Πρόσθεση πινάκων 39

6.2.3 Πολλαπλασιασμός πινάκων 40

6.2.4 Μετάθεση πινάκων 41

6.2.5 Παράλληλο πρόγραμμα αθροίσματος στοιχείων πίνακα. 43

6.3 Επιπλέον Μετρήσεις 44

6.3.1 Διαφορετική διεύθυνση για την συνάρτηση _mm_prefetch. 45

6.3.2 Διαφορετική παράμετρο συνάρτηση _mm_prefetch. 45

6.3.4 Διαφορετική παράμετρο για την εντολή της assembly. 45

6.4 Επιπλέον Γραφικές Παραστάσεις 46

6.4.1 Διαφορετική διεύθυνση για την συνάρτηση _mm_prefetch. 46

6.4.2 Διαφορετική παράμετρο συνάρτηση _mm_prefetch. 48

6.4.4 Διαφορετική παράμετρο για την εντολή της assembly. 50

ΚΕΦΑΛΑΙΟ **7** **Συμπεράσματα** …….. **52**

7.1 Συμπεράσματα 52

7.2 Μελλοντική εργασία 53

Β ι β λ ι ο γ ρ α φ ί α ……. **54**

Π α ρ ά ρ τ η μ α Α…….……...…….…… **56**

Π α ρ ά ρ τ η μ α Β…… **64**

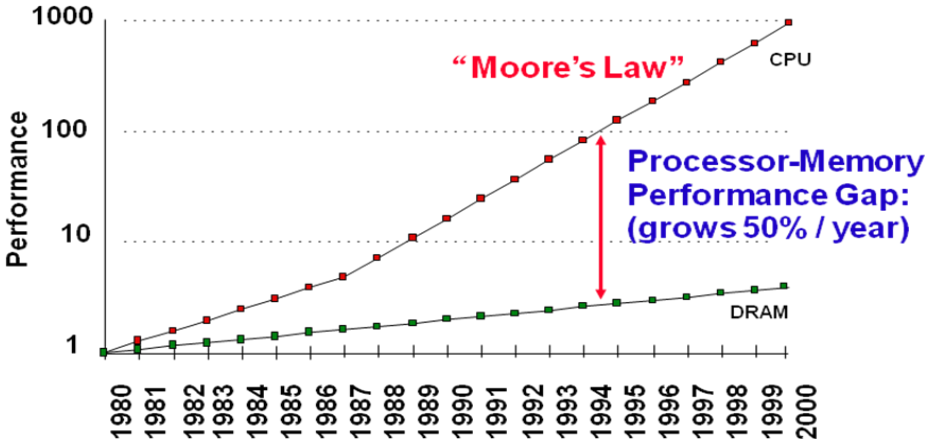
Κεφάλαιο 1

Εισαγωγή

1.1 Γενική εισαγωγή στο Prefetching 9
1.2 Το μοντέλο DDM 10

1.1 Γενική εισαγωγή στο Prefetching

Η απόδοση των επεξεργαστών έχει μεγαλώσει με πολύ πιο γρήγορο ρυθμό από την απόδοση της μνήμης τις τελευταίες τρεις δεκαετίες. Καθώς η απόδοση των επεξεργαστών ακολούθησε τον νόμο του Moore's και βελτιώθηκε κατά 50 % ετησίως μέχρι το 2004 και 20% από τότε, ενώ η απόδοση της μνήμης βελτιώθηκε κατά 9 % κάθε χρόνο. Αυτό προκάλεσε ένα μεγάλο χάσμα μεταξύ τους και έκανε την μνήμη να αποτελεί σημαντικό bottleneck της απόδοσης. Το πρόβλημα αυτό χειροτερεύει με την ταχεία ανάπτυξη των πολυπύρηνων επεξεργαστών, διότι στους πολλαπλούς πυρήνες η μνήμη διαμοιράζεται μεταξύ τους. Παραδοσιακά οι μνήμες cache χρησιμοποιούνταν για την βελτίωση της απόδοσης των προσβάσεων της μνήμης χρησιμοποιώντας την αρχή της τοπικότητας. Εν τούτης, οι βελτιστοποιήσεις είναι απαραίτητες για την βελτίωση της χρήσης των μνήμων cache και για την ελαχιστοποίηση των cache misses.



Το prefetching δεδομένων θεωρείται ένας αποτελεσματικός τρόπος να για να συγκαλύψει την καθυστέρηση στην πρόσβαση των δεδομένων που προκαλείται από cache misses και να γεφυρώσει το χάσμα απόδοσης μεταξύ του επεξεργαστή και της μνήμης. Το prefetching δεδομένων είναι μια τεχνική απόκρυψης της καθυστέρησης της πρόσβασης των δεδομένων, που αποσυνδέει και επικαλύπτει τις μεταφορές δεδομένων και τον υπολογισμό. Προκειμένου να μειώσουμε την στασιμότητα του CPU σε ένα cache miss, το prefetching δεδομένων προβλέπει τις μελλοντικές προσβάσεις δεδομένων, ξεκινά μια προσκόμιση δεδομένων, και φέρνει τα δεδομένα πιο κοντά στον επεξεργαστή πριν να ζητηθούν από αυτόν.

1.2 Το μοντέλο DDM

Το DDM είναι ένα non-blocking multithreading μοντέλο που συνδυάζει τα πλεονεκτήματα ενός Data Flow μοντέλου στην αξιοποίηση ταυτοχρονισμού με την υψηλή απόδοση της σειριακής επεξεργασίας των βασικών μικροεπεξεργαστών. Επιπλέον, το DDM μπορεί να βελτιώσει την τοπικότητα της σειριακής επεξεργασίας με την ενσωμάτωση prefetching ντετερμινιστικών δεδομένων, χρησιμοποιώντας data-driven caching policies. Ο πυρήνας της υλοποίησης του DDM (Thread Synchronization Unit -μια μονάδα υλικού που εφαρμόζεται απευθείας πάνω στον δίαυλο του επεξεργαστή) είναι υπεύθυνος για την χρονοδρομολόγηση των threads κατά τον χρόνο εκτέλεσης βασιζόμενος στην διαθεσιμότητα των δεδομένων.
Στο DDM ένα πρόγραμμα είναι μια συλλογή από code-blocks (αντίστοιχο ενός loop, function). Ανά πάσα χρονική στιγμή μόνο μερικά code-blocks βρίσκονται μέσα στο TSU.

Κάθε code-block είναι μια συλλογή από threads, τα οποία με την σειρά τους είναι μια συλλογή από εντολές. Τα threads μπαίνουν σε μια σειρά producer-consumer και ένα thread εκτελείται μόνο αν τα δεδομένα του είναι έτοιμα. Η σειρά εκτέλεσης των εντολών ενός thread εναπόκειται στο CPU (σειριακά ή out-of-order).

Σε ένα DDM μοντέλο κάθε thread αναγνωρίζεται από το thread number (Thread#) που αποτελείται από την εξής πλειάδα:

- Context που καθορίζεται κατά το runtime (dynamic) και χρησιμοποιείται για να ξεχωρίζει διαφορετικές κλήσεις του ίδιου code block ή thread.
- Block που προσδιορίζει το code block.
- ThreadID, που είναι static και προσδιορίζει το thread μέσα στο code block.

Το synchronization template του κάθε thread περιέχει τις ακόλουθες πληροφορίες:

1. Το Instruction Frame Pointer (IFP): δείκτης στην διεύθυνση της πρώτης εντολής του thread.

2. Το ReadyCount (RC): τιμή που δείχνει τον αριθμό των παραγωγών-threads που κάποιο thread περιμένει για να μπορέσει να αρχίσει την εκτέλεση του. Ένα thread είναι έτοιμο για εκτέλεση όταν το ReadyCount του ισούται με 0.

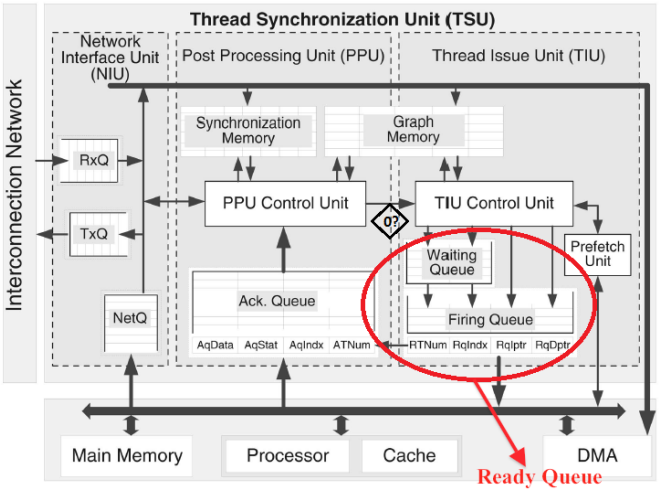
3. Το Data Frame Pointer (DFP): δείκτης στα πλαίσια δεδομένων που έχουν ανατεθεί σε ένα thread/code block. Μας επιτρέπει επαναχρησιμοποίηση code blocks με διαφορετικές τιμές δεδομένων.

4. Τα Consumer Threads (Consumer1 and Consumer2): είναι οι καταναλωτές ενός thread. Καθορίζουν ποιών threads οι τιμές του ReadyCount θα μειωθούν κατά 1 μετά από την ολοκλήρωση της εκτέλεσης του thread. Αν το thread έχει μόνο ένα Consumer τότε το Consumer2 ισούται με 0. Αν έχει περισσότερα από δύο τότε το Consumer1 ισούται με 0 και το Consumer2 δείχνει σε μια λίστα με τους Consumers. Ο λόγος που υπάρχουν μόνο δύο Consumers οφείλεται στο γεγονός πως οι πλειονότητα των DDM threads έχει ένα ή δύο Consumers.

1.2.1 Το Thread Synchronization Unit

Όπως φαίνεται από το σχήμα 1, το Graph Memory (GM) και το Synchronization Memory (SM) αποτελούν τις δύο κύριες αποθηκευτικές μονάδες του TSU.

- Graph Memory (GM): περιέχει το IFP, DFP και τους καταναλωτές κάθε thread.
- Synchronization Memory (SM): περιέχει το ReadyCount για κάθε thread.



Σχήμα 1 – TSU

Η επικοινωνία μεταξύ του επεξεργαστή υπολογισμού και του TSU επιτυγχάνεται με χρήση δύο ουρών, Ready Queue (RQ) και Acknowledgement Queue (AQ). Οι δύο αυτές ουρές είναι memory mapped, γεγονός που μας απαλλάσσει από περισσότερη πολυπλοκότητα (τροποποίηση του επεξεργαστή ή προσθήκη extra εντολών). Η RQ περιέχει δείκτες (IFP, DFP) στα threads που είναι έτοιμα για εκτέλεση. Όπως φαίνεται και από το πιο πάνω σχήμα το RQ χωρίζεται σε δύο διαφορετικές ουρές:

- Waiting Queue (WQ): η πρώτη ουρά στην οποία μπαίνουν τα έτοιμα threads μαζί με το Thread# και το Iteration number τους.
- Firing Queue (FQ): η δεύτερη και τελευταία ουρά στην οποία μπαίνουν τα έτοιμα threads μαζί με τα στοιχεία RTNum, RqIndx, RqIptr, RqDptr, τα οποία αντλούνται από το Graph Memory χρησιμοποιώντας το Thread# από το WQ.

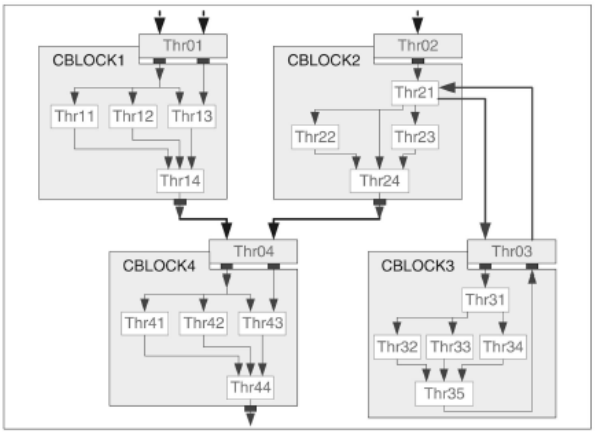
Το AQ περιέχει τις πληροφορίες και την κατάσταση των threads που έχουν εκτελεστεί.

Όπως φαίνεται από το σχήμα 1 το Thread Synchronization Unit (TSU) αποτελείται από τρεις μονάδες:

- Post Processing Unit (PPU), το οποίο διαβάζει από το AQ όσα threads έχουν εκτελεστεί από τον επεξεργαστή. Ακολούθως, ανακαλύπτει μέσω του GM τους καταναλωτές αυτών των threads και ενημερώνει το ReadyCount τους μέσα στο SM. Αν το ReadyCount κάποιου thread ισούται με 0, τότε το προωθεί στο TIU.
- Thread Issue Unit (TIU), του οποίου η κύρια λειτουργικότητα είναι η δρομολόγηση των threads που θεωρούνται έτοιμα από το PPU. Ουσιαστικά εισάγει τα έτοιμα threads μέσα στο WQ, τα οποία στη συνέχεια μεταφέρονται στο FQ. Ακολούθως, ο επεξεργαστής έρχεται και διαβάζει τη διεύθυνση του επόμενου προς εκτέλεση thread από το FQ.
- Network Interface Unit (NIU), το οποίο είναι υπεύθυνο για την επικοινωνία μεταξύ του TSU και του διασυνδεδεμένου δικτύου. Αν για παράδειγμα ένα thread είναι έτοιμο για εκτέλεση, αλλά ανήκει σε έναν απομακρυσμένο κόμβο, το thread αυτό προωθείτε στο NIU για περεταίρω επεξεργασία.

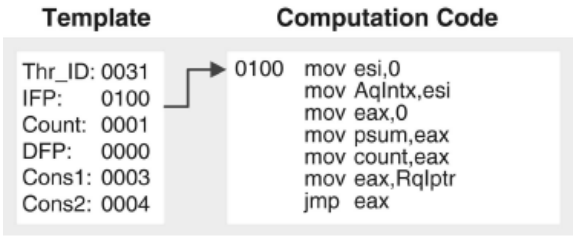
1.2.2 Το Codeblock

Στο σχήμα 2 φαίνεται η δρομολόγηση των code blocks. Κάθε code block συσχετίζεται με ένα ειδικό thread, το λεγόμενο inlet thread (Thr01, Thr02, Thr03, Thr04), το οποίο χρησιμοποιείται για να παρέχει συγχρονισμό ανάμεσα στα code blocks. Το ReadyCount ενός inlet thread ισούται με τον αριθμό των εισόδων στο code block. Όπως και τα υπόλοιπα threads, έτσι και το inlet thread είναι έτοιμο για εκτέλεση όταν το ReadyCount στο SM ισούται με 0. Τα inlet threads είναι υπεύθυνα για την φόρτωση των προτύπων του code block στο GM, την αρχικοποίηση του SM με βάση τα ReadyCount του κάθε thread και ενημέρωση του DFP στο GM. Εκτός από inlet thread, κάθε code block έχει και ένα outlet thread το οποίο απομακρύνει το code block από το TSU και απελευθερώνει την μνήμη που είχε δεσμευτεί για αυτό.



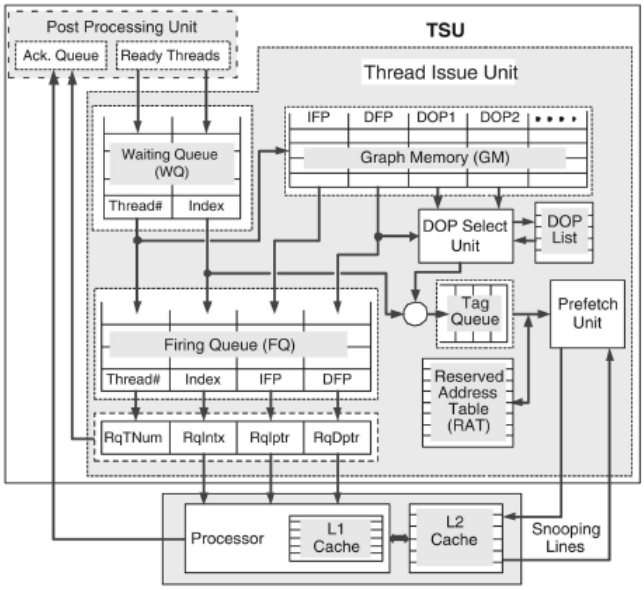
Σχήμα 2 – Code block Scheduling

Στο σχήμα 3 φαίνεται ο κώδικας ενός thread. Ο πρώτος αριθμός είναι ο αριθμός του thread. Ο δεύτερος είναι η διεύθυνση εκκίνησης του κώδικα του thread. Ο τρίτος αριθμός αφορά των αριθμό των εισόδων στο thread, ο τέταρτος το DFP το οποίο αρχικοποιείται με 0 και τέλος, οι τελευταίοι δύο είναι οι αριθμοί των consumer threads.



Σχήμα 3 – Code of a thread

1.2.3 Cacheflow



Σχήμα 4 – TIU with false conflict avoidance CacheFlow support

Αν και η δρομολόγηση με βάση την διαθεσιμότητα των δεδομένων μπορεί να έχει αρνητικά αποτελέσματα στην τοπικότητα εντούτοις, μπορούμε να την εκμεταλλευτούμε και να δημιουργήσουμε αποδοτικές πολιτικές διαχείρισης μνήμης cache οι οποίες μειώνουν τα cache misses. Αυτές οι πολιτικές ονομάζονται CacheFlow Policies.
Η διάσπαση του RQ σε δύο ουρές (WQ και FQ) αποσκοπεί στην υλοποίηση των CacheFlow Policies. Αυτό επιτυγχάνεται με την αρχική τοποθέτηση των έτοιμων threads μέσα στο WQ και έπειτα με τον καθορισμό των offset διευθύνσεων αυτών των threads από το TSU και την προ-ανάκληση των αντίστοιχων δεδομένων, πριν αυτά τα threads εισέρθουν στο FQ.
Ωστόσο, αυτή η προ-ανάκληση των δεδομένων μπορεί να προκαλέσει false cache conflicts (για παράδειγμα, πριν ένα thread να εκτελεστεί ένα από τα προ-ανακαλούμενα cache μπλοκ του αντικαθιστάται από ένα άλλο cache μπλοκ ενός άλλου thread). Η πιθανότητα εμφάνισης false cache conflicts μπορεί να μειωθεί με το να κρατάμε το μέγεθος της FQ όσο πιο μικρό γίνεται.

Για το πρόβλημα αυτό υπάρχει βελτιστοποίηση με όνομα false conflict avoidance (Σχήμα 4), η οποία αποτρέπει τον prefetcher από το να αντικαταστήσει cache μπλοκ τα οποία χρειάζονται τα threads που βρίσκονται ήδη μέσα στο FQ. Αυτό επιτυγχάνεται με την χρησιμοποίηση ενός πίνακα και μιας ουράς:

- Reserved Address Table (RAT), ο οποίος περιέχει τις διευθύνσεις όλων των cache μπλοκ τα οποία χρειάζονται τα threads που βρίσκονται μέσα στο FQ, καθώς και το thread που τρέχει.
- Tag Queue, η οποία περιέχει όλες τις διευθύνσεις cache μπλοκ τις οποίες χρειάζεται ένα έτοιμο thread που βρίσκεται στο WQ.

Οι διευθύνσεις των πιο πάνω στοιχείων συγκρίνονται μεταξύ τους για να φανεί κατά πόσον το prefetching μπορεί να προκαλέσει false cache conflicts (αν υπάρχουν κοινές διευθύνσεις μέσα στα δύο στοιχεία). Αν δεν υπάρχει πιθανότητα να συμβεί false cache conflict το έτοιμο thread μετακινείται μέσα στο FQ, αλλιώς τοποθετείται προσωρινά μέσα σε ένα buffer και το επόμενο έτοιμο thread στο WQ ελέγχετε. Τα threads που βρίσκονται μέσα στο buffer έχουν προτεραιότητα έναντι των threads που βρίσκονται μέσα στο WQ, έτσι ώστε να μην υπάρχει κίνδυνος παρατεταμένης στέρησης (μπλοκάρει τους consumers του).
Μια άλλη βελτιστοποίηση αφορά την προσπάθεια για εκμετάλλευση της τοπικότητας με την αναδιάταξη των threads που βρίσκονται μέσα στο WQ. Αυτή η βελτιστοποίηση ονομάζεται thread reordering. Με την τεχνική αυτή τα threads που έχουν ίδιο Thread# τοποθετούνται το ένα δίπλα στο άλλο μέσα στο WQ και παράλληλα, ταξινομούνται με βάση το index τους.

Και οι δύο πιο πάνω μηχανισμοί (thread reordering και false conflict avoidance) λειτουργούν παράλληλα και ασύγχρονα με το υπόλοιπο TIU και γι' αυτό δεν προκαλούν οποιεσδήποτε επιπλέον καθυστερήσεις.

Κεφάλαιο 2

Prefetching

2.1 Θέματα Prefetching 17
2.2 Hardware και Software Prefetching 19

2.1 Θέματα Prefetching

Τώρα, τι είναι απαραίτητο ώστε να σχεδιάσουμε μια στρατηγική για data prefetching. Κατά παράδοση, το prefetching αφορά τρία θέματα: τι να κάνουμε prefetch, πότε να κάνουμε prefetch και πού να κάνουμε prefetch. Το τί να κάνουμε prefetch αποφασίζει ποια δεδομένα μπορεί ο επεξεργαστής να χρειαστεί στο μέλλον. Το πότε να κάνουμε prefetch αποφασίζει πόσο νωρίς πρέπει να φέρουμε τα δεδομένα και να αποφύγουμε οποιαδήποτε αρνητικά αποτελέσματα λόγω του prefetching. Το πού να κάνουμε prefetch αποφασίζει τον προορισμό των δεδομένων που έχουμε κάνει prefetch. Οι περισσότερες υπάρχον μέθοδοι συγκεντρώνονται στο τί να κάνουμε prefetch. Αν λάβουμε υπόψη το γεγονός ότι οι πολυπύρηνοι επεξεργαστές έχουν καθιερωθεί πλέον, το prefetching μπορεί να είναι μια ισχυρή τεχνική για την επίλυση της καθυστέρησης της πρόσβασης των δεδομένων.

2.1.1 Τί να κάνουμε Prefetch

Το να προβλέπουμε του τί δεδομένα να κάνουμε Prefetch είναι η πιο σημαντική προϋπόθεση του prefetching. Εάν μια στρατηγική prefetching μπορεί να προβλέψει την εμφάνιση μελλοντικών cache misses, τότε μπορούν εντολές για prefetching να εκδοθούν πιο πριν και να φέρουν τα δεδομένα πριν από τη στιγμή που έχουμε cache misses. Για να αποκρύψουμε την καθυστέρηση που προκαλείται από τα cache misses αποτελεσματικά, πρέπει η ακρίβεια

της πρόβλεψης του τι να κάνουμε prefetch να είναι υψηλή. Η πρόβλεψη μελλοντικών αναφορών δεδομένων είναι κρίσιμη. Χαμηλή ακρίβεια οδηγεί σε cache pollution.

2.1.2 Πότε να κάνουμε Prefetch

Η στιγμή που θα εκδώσουμε μια εντολή prefetch έχει σημαντική επίδραση στην συνολική απόδοση του prefetching. Τα δεδομένα που κάναμε Prefetched πρέπει να φτάσουν στον προορισμό τους πριν να συμβεί ένα raw cache miss. Η αποδοτικότητα του έγκαιρου prefetching εξαρτάται από το συνολικό prefetching overhead (το overhead της πρόβλεψης μελλοντικών προσβάσεων και επιπλέον το overhead στο prefetching δεδομένων) και το χρόνο για την εμφάνιση του επόμενου cache miss. Εάν το συνολικό prefetching overhead υπερβαίνει το χρόνο του επόμενου cache miss, η προσαρμογή του διαστήματος του prefetching μπορεί να βοηθήσει στην αποφυγή των αργών prefetches.
Τα prefetches πρέπει να εκδοθούν κατά κάποιο χρονικό τρόπο. Εάν ένα prefetch εκδοθεί πολύ νωρίς τότε υπάρχει μια πιθανότητα ότι τα prefetched δεδομένα θα εκτοπίσουν άλλα χρήσιμα δεδομένα από τα πιο υψηλά επίπεδα της ιεραρχίας της μνήμης ή θα εκτοπιστούν αυτά πριν από την χρήση τους. Εάν ένα prefetch εκδοθεί πολύ αργά, μπορεί να μην φτάσει πριν από την πραγματική του αναφορά από την μνήμη και ως εκ τούτου να εισάγει κύκλους καθυστέρησης στον επεξεργαστή.

2.1.3 Πού να κάνουμε Prefetch

Η απόφαση του πού να τοποθετήσουμε τα δεδομένα που κάναμε prefetch στην ιεραρχία μνήμης είναι μια βασική απόφαση σχεδιασμού. Σαφώς, τα δεδομένα πρέπει να μεταφερθούν σε ένα υψηλό επίπεδο της ιεραρχίας της μνήμης ούτως ώστε να προσφέρουν ένα σημαντικό όφελος στην απόδοση. Η πλειοψηφία των συστημάτων τοποθετούν τα prefetched δεδομένα σε κάποιου τύπου μνήμη cache. Τα δεδομένα μπορούν να γίνουν prefetched σε μια μνήμη cache που είναι τοπική στον επεξεργαστή ή σε μια μνήμη cache που μοιράζεται από πολλαπλούς πυρήνες, ή σε ένα ξεχωριστό prefetch cache. Ένα ξεχωριστό prefetch cache μπορεί να είναι είτε private σε ένα πυρήνα είτε να μοιράζεται από πολλαπλούς πυρήνες.

2.2 Hardware και Software Prefetching

Το Hardware Prefetching μπορεί να χρειάζεται πολύπλοκο και ακριβό Hardware, ενώ το software prefetching χρειάζεται επιπλέον CPU εντολές. Ενώ το software prefetching απαιτεί λιγότερη υποστήριξη hardware, πρέπει να δημιουργήσει τις εντολές υπολογισμού διεύθυνσης και μια εντολή prefetching για κάθε δεδομένο που πρέπει να κάνει prefetched. Το Hardware Prefetching, ωστόσο, πρέπει να γίνεται σταδιακά όλο και πιο περίπλοκο για να είναι σε θέση να υπολογίσει τα βήματα της πρόσβασης δεδομένων και για να αυξησεί το prefetching lookahead.

Ακόμη το Software Prefetching έχει μεγαλύτερο μέγεθος κώδικα και μπορεί να ανιχνεύσει στατικά μοτίβα, ενώ το Hardware Prefetching μπορεί να ανιχνεύσει δυναμικά μοτίβα. Επιπλέον το software prefetching είναι συνήθως πολύ ακκριβές, αλλά του αναλογεί κάποιο runtime overhead και δεν μπορεί να εκδώσει εντολές prefetching αρκετά νωρίς, ούτως ώστε να κρύψει την καθυστέρηση που δημιουργείται από την πρόσβαση στην κύρια μνήμη. Το Hardware Prefetching μπορεί να κάνει prefetch χωρικές περιοχές, αλυσίδες δεικτών, ή επαναεμφανιζόμενων μοτίβων. Ενώ αυτά μπορούν να αποκρύψουν αρκετό από το χρόνο πρόσβασης της μνήμης, μπορούν επιπρόσθετα να καταναλώσουν αρκετά σημαντικά ποσά του bandwidth της μνήμης.

Κεφάλαιο 3

Cache memory

3.1 Η μνήμη Cache 20
3.2 Cache Associativity 21
3.3 Cache Entries 23
3.4 Απόδοσης της μνήμης Cache 23
3.5 Cache Misses 24
3.6 Cache pollution και eviction 25
3.7 Inclusive Cache και Exclusive cache 26

3.1 Η μνήμη Cache

Η μνήμη cache είναι μια μνήμη τυχαίας προσπέλασης που είναι ενσωματωμένη στην κεντρική μονάδα επεξεργασίας ενός υπολογιστή, ή βρίσκεται δίπλα της σε ένα ξεχωριστό τσιπ . Χρησιμοποιείται κυρίως για την μείωση του χρόνου που χρειαζόμαστε για την πρόσβαση της μνήμης. Η μνήμη cache είναι μια πιο μικρή και πιο γρήγορη μνήμη που αποθηκεύει αντίγραφα των δεδομένων από τις πιο συχνά χρησιμοποιημένες θέσεις της κύριας μνήμης. Εάν οι περισσότερες προσβάσεις μνήμης είναι αποθηκευμένες στην cache, ο μέσος όρος του χρόνου αναμονής (latency) των προσβάσεων μνήμης θα είναι πιο κοντά στον μέσο όρο του χρόνου αναμονής (latency) της cache παρά της κύριας μνήμης. Όταν ο επεξεργαστής πρέπει να διαβάσει ή να γράψει από μια θέση στην κύρια μνήμη, πρώτα ελέγχει αν υπάρχει ένα αντίγραφο του στη μνήμη cache. Αν ναι, ο επεξεργαστής διαβάζει ή γράφει αμέσως από την cache, το οποίο και είναι πολύ πιο γρήγορο από το να διαβάζει ή να γράφει από την κύρια μνήμη.

Η μνήμη cache συνήθως περιγράφεται σε επίπεδα στενότητας και προσβασιμότητας στο μικροεπεξεργαστή. Η μνήμη L1 cache είναι στο ίδιο τσιπ, όπως ο μικροεπεξεργαστής ενώ η μνήμη L2 cache είναι συνήθως ένα ξεχωριστό τσιπ στατικής RAM (SRAM).

3.2 Cache Associativity

Το CPU cache associativity καθορίζει τον τρόπο που τα δεδομένα της cache συσχετίζονται με τις θέσεις στην κύρια μνήμη. Η πολιτική αντικατάστασης αποφασίζει πού στη μνήμη cache ένα αντίγραφο μιας συγκεκριμένης καταχώρησης της κύριας μνήμης θα πάει. Αν η πολιτική αντικατάστασης είναι ελεύθερη να επιλέξει οποιαδήποτε θέση στη μνήμη cache για να κρατήσει το αντίγραφο, η μνήμη cache ονομάζεται fully associative. Από την άλλη, αν κάθε εγγραφή στην κύρια μνήμη μπορεί να πάει σε μία μόνο θέση στη μνήμη cache, η μνήμη ονομάζεται direct mapped. Πολλές μνήμες cache εφαρμοζουν ένα συμβιβασμό στον οποίο η κάθε καταχώρηση στην κύρια μνήμη μπορεί να πάει σε οποιαδήποτε από N θέσεις στη μνήμη cache, και περιγράφονται ως σύνολο N-way set associative.

Το Associativity είναι ένα trade-off. Αν υπάρχουν δέκα θέσεις στις οποίες η πολιτική αντικατάστασης θα μπορούσε να χαρτογραφήσει μια θέση μνήμης, τότε για να ελέγξουμε αν η θέση είναι στη μνήμη cache, πρέπει να αναζητηθεί στις δέκα αυτές καταχωρήσεις της μνήμης cache .Ο έλεγχος περισσότερων θέσεων χρειάζεται περισσότερη ενέργεια και χρόνο. Από την άλλη πλευρά, μνήμες cache με μεγαλύτερο associativity έχουν λιγότερα misses , έτσι ώστε το CPU σπαταλά λιγότερο χρόνο διαβάζοντας από την αργή κύρια μνήμη. Ο κανόνας είναι ότι ο διπλασιασμός του associativity, από direct mapped σε 2-way, ή από 2-way σε 4-way, έχει περίπου το ίδιο αποτέλεσμα στο hit rate, όπως ο διπλασιασμός του μεγέθους της cache. Η αύξηση του Associativity πέρα από 4-way έχει όμως πιο μικρή επίδραση στο hit rate.

3.2.1 Direct Mapped

Ένα μπλοκ μνήμης cache μπορεί να πάει μόνο σε μία θέση στην cache. Έτσι είναι πολύ εύκολο να βρεθεί ένα μπλοκ μνήμης, αλλά δεν είναι πολύ ευέλικτο για το πού να τοποθετήσει το μπλοκ. Αυτό σημαίνει ότι εάν δύο θέσεις αντιστοιχίζονται στην ίδια καταχώρηση, μπορεί να συγκρούονται το ένα με το άλλο συνεχώς. Αν και είναι πιο απλή, μια direct-mapped cache πρέπει να

είναι πολύ μεγαλύτερη από μια associative για να παρέχει συγκρίσιμη επίδοση,και είναι πιο απρόβλεπτη.
Έτσι, αν έχουμε 64 MB διευθύνσεις κύριας μνήμης, κάθε γραμμή της cache θα μοιράζεται από 4.096 διευθύνσεις μνήμης (64 M διαιρούμενο με 16 K).

3.2.2 2-Way Set Associative

Αυτή η μνήμη cache αποτελείται από σετ (sets) που μπορούν να χωρέσουν δύο μπλοκ στο καθένα. Το index χρησιμοποιείται τώρα για να βρεθεί το σετ, και το tag βοηθά για να βρεθεί το μπλοκ μέσα στο σετ.
Ένα από τα οφέλη αυτού είναι ότι τα tags που είναι αποθηκευμένα στη μνήμη cache δεν πρέπει να περιλαμβάνουν το μέρος της κύριας διεύθυνσης της μνήμης που υπονοείται με το index της μνήμης cache του. Δεδομένου τα tags της μνήμης cache έχουν λιγότερα bits, παίρνουν λιγότερο χώρο στο chip του μικροεπεξεργαστή και μπορούν να διαβαστούν και να συγκριθούν ταχύτερα.

3.2.3 4-Way Set Associative

Κάθε σετ εδώ χωράει τέσσερα μπλοκ, έτσι υπάρχουν λιγότερα σύνολα. Ως εκ τούτου, απαιτούνται λιγότερα index bits.
Έτσι, αντί για ένα ενιαίο μπλοκ των 16.384 γραμμών, έχουμε 4.096 σετ με 4 γραμμές σε καθένα. Κάθε ένα από αυτά τα σετ μοιράζεται από 16.384 διευθύνσεις μνήμης (64 M διαιρείται με 4 K) αντί των 4.096 διευθύνσεων στην περίπτωση της direct mapped μνήμης cache. Έτσι υπάρχουν περισσότερα για να διαμοιραστούν (4 γραμμές αντί για 1), αλλά περισσότερες διευθύνσεις που τα μοιράζονται (16.384 αντί του 4096).

3.2.4 Fully Associative

Δεν χρειάζεται index, δεδομένου ότι ένα μπλοκ μνήμης cache μπορεί να τοποθετηθεί οπουδήποτε στην cache. Κάθε tag πρέπει να συγκρίνεται όταν βρεθεί ένα μπλοκ στην μνήμη cache, αλλά η τοποθέτηση των μπλοκ, είναι πολύ ευέλικτη. Η fully associative cache έχει το καλύτερο hit ratio, διότι οποιαδήποτε γραμμή στη μνήμη cache μπορεί να κρατήσει οποιαδήποτε διεύθυνση που πρέπει να είναι αποθηκευμένη στην cache. Το μειονέκτημα της είναι όμως ότι είναι συνήθως πιο αργή και περίπλοκη.

3.3 Cache Entries

Η μνήμη χωρίζεται σε cache "γραμμές". Κάθε πρόσβαση σε δεδομένα που αφορούν τη μνήμη cache χρησιμοποιεί αυτό το μέγεθος, το οποίο τείνει να είναι μεγαλύτερο από το μεγαλύτερο μέγεθος κάποιου αιτήματος της CPU.
Κάθε θέση μνήμης μπορεί να προσδιοριστεί από μια φυσική διεύθυνση μνήμης. Όταν η μνήμη έχει αντιγραφεί στη μνήμη cache, μια καταχώρηση μνήμης cache δημιουργείται (cache entry). Μπορεί να περιλαμβάνει την αιτούμενη θέση μνήμης (tag) και ένα αντίγραφο των δεδομένων.

Όταν ο επεξεργαστής χρειάζεται να διαβάσει ή να γράψει σε μια θέση στην κύρια μνήμη, πρώτα ελέγχει για μια αντίστοιχη καταχώρηση στη μνήμη cache. Η μνήμη cache ελέγχει για το περιεχόμενο της αιτούμενης θέση μνήμης σε κάθε γραμμή της cache που μπορεί να περιέχει αυτή τη διεύθυνση. Αν ο επεξεργαστής διαπιστώσει ότι η θέση μνήμης είναι στη μνήμη cache, ένα cache hit έχει συμβεί αλλιώς, έχει συμβεί ένα cache miss.
Σε περίπτωση cache hit , ο επεξεργαστής διαβάζει ή γράφει αμέσως τα δεδομένα στη γραμμή cache. Σε περίπτωση cache miss, η μνήμη cache διαθέτει μια νέα εγγραφή, και αντίγραφει τα δεδομένα από την κύρια μνήμη. Στη συνέχεια, το αίτημα ικανοποιείται από τα περιεχόμενα της μνήμης cache.

3.4 Απόδοση της μνήμης Cache

Το ποσοστό των προσβάσεων που έχουν ως αποτέλεσμα ένα cache hit είναι γνωστό ως το ποσοστό επιτυχίας (hit rate), και μπορεί να είναι ένα μέτρο της αποτελεσματικότητας της μνήμης cache για ένα συγκεκριμένο πρόγραμμα ή αλγόριθμο.
Από την άλλη τα Read misses καθυστερήσουν την εκτέλεση, γιατί χρειάζονται τα δεδομένα να μεταφέρονται από μνήμη πολύ πιο αργή από την ίδια την μνήμη cache. Τα Write misses μπορούν να συμβούν χωρίς μια τέτοια ποινή, δεδομένου ότι ο επεξεργαστής μπορεί να συνεχίσει την εκτέλεση ενόσω στο παρασκήνιο τα δεδομένα αντιγράφονται στην κύρια μνήμη.

3.5 Cache Misses

Ένα cache miss αναφέρεται σε μια αποτυχημένη προσπάθεια να διαβαστεί ή να γραφτεί ένα κομμάτι των δεδομένων στη μνήμη cache, το οποίο οδηγεί σε μια πρόσβαση στη κύρια μνήμη με πολύ μεγαλύτερη καθυστέρηση (latency). Υπάρχουν τρία είδη μνήμης cache misses: instruction read miss, data read miss, kai data write miss.

Ένα cache read miss από μία instruction cache γενικά προκαλεί την πιο μεγάλη καθυστέρηση, επειδή ο επεξεργαστής, ή τουλάχιστον το νήμα της εκτέλεσης, θα πρέπει να περιμένει (stall), έως ότου η εντολή τραβηχτεί από την κύρια μνήμη. Ένα cache read miss από μία data cache συνήθως προκαλεί μικρότερη καθυστέρηση, επειδή οι εντολές που δεν εξαρτούνται από την ανάγνωση της μνήμης cache μπορεί να εκδοθούν και να συνεχίσουν την εκτέλεση τους μέχρι την επιστροφή των δεδομένων από την κύρια μνήμη και μέχρι οι εξαρτώμενες εντολές μπορούν να συνεχίσουν την εκτέλεση. Ένα cache read miss από μία data cache γενικά προκαλεί τη μικρότερη καθυστέρηση, διότι η εγγραφή μπορεί να μπει σε ουρά και επειδή υπάρχουν λίγοι περιορισμοί σχετικά με την εκτέλεση των επόμενων εντολών. Ο επεξεργαστής μπορεί να συνεχίσει έως ότου η ουρά είναι γεμάτη.

3.5.1 Compulsory misses

Compulsory misses είναι τα misses που προκαλούνται από την πρώτη αναφορά σε μια θέση στη μνήμη. Το μέγεθος της μνήμης cache και το associativity της δεν κάνουν καμία διαφορά στον αριθμό των Compulsory misses. Το Prefetching μπορεί να βοηθήσει εδώ, όπως και μεγαλύτερα μεγέθη μπλοκ μνήμης cache (που είναι μια μορφή prefetching). Τα Compulsory misses κάποτε αναφέρονται και ως cold misses. Ωστόσο, δεδομένου ότι ένα compulsory miss συμβαίνει μόνο όταν τα δεδομένα φορτώνονται την πρώτη φορά, σε μακροπρόθεσμη βάση με ένα πρόγραμμα με πολλές προσβάσεις μνήμης, τα Compulsory misses δεν είναι πρόβλημα.

3.5.2 Capacity misses

Τα Capacity misses είναι τα misses που συμβαίνουν ανεξάρτητα από το associativity ή το μέγεθος του μπλοκ, αλλά οφείλεται αποκλειστικά στο πεπερασμένο μέγεθος της μνήμης

cache. Ένα capacity miss συμβαίνει όταν μια θέση μνήμης προσπελαστεί μια φορά, αλλά αργότερα, διότι η μνήμη έχει γεμίσει, τα δεδομένα αυτά απορρίπτονται, και στη συνέχεια, όταν παίρνουμε ένα miss όταν έχουμε και πάλι πρόσβαση σε αυτή τη θέση μνήμης , επειδή τα δεδομένα δεν είναι πλέον στην μνήμη. Η λύση σε αυτό είναι να αυξηθεί το μέγεθος της μνήμης cache. Σημειώστε ότι δεν υπάρχει καμία χρήσιμη έννοια της μνήμης cache αν είναι "πλήρης" ή"άδεια" ή "μισογεμάτη". Οι μνήμες cache έχουν σχεδόν πάντα κάθε γραμμή που γεμάτη με ένα αντίγραφο κάποιας γραμμής της κύρια μνήμης, και σχεδόν κάθε διάθεσης μιας νέας γραμμής απαιτεί την έξωση μιας παλαιάς γραμμής (cache eviction).

3.5.3 Conflict misses

Τα Conflict misses είναι τα misses που θα μπορούσαν να είχαν αποφευχθεί, αν δεν γινόταν από την cache έξωση μια καταχώρησης νωρίτερα. Ένα conflict miss συμβαίνει όταν υπάρχουν πολλές προσβάσεις μνήμης που αντιστοιχούν στο ίδιο index σετ σε μια cache, έτσι τα δεδομένα σε ένα μπλοκ του σε αυτό το index σετ μπορεί να απορρίπτεται ακόμη και αν ένα άλλο μπλοκ σε άλλο σχετικά αχρησιμοποίητο index σετ μπορεί να είναι ακόμη πιο παλιό. Αν αυτό το μπλοκ, απορρίπτεται και στη συνέχεια ανακτάται, αυτό ονομάζεται conflict miss. Μπορεί κάπως να αποφευχθεί εαν αυξηθεί το associativity.

3.6 Cache pollution και eviction

Το Cache pollution περιγράφει τις καταστάσεις στις οποίες ένα εκτελέσιμο πρόγραμμα του υπολογιστή φορτώσει δεδομένα στην cache του CPU άσκοπα, προκαλώντας έτσι σε άλλα απαιτούμενα δεδομένα να εκδιωχθούν από τη μνήμη cache σε χαμηλότερα επίπεδα της ιεραρχίας της μνήμης, πιθανώς και τελείως κάτω στην κύρια μνήμη, προκαλώντας έτσι μια πτώση της απόδοσης. Έχουν γίνει προσπάθειες να μειωθεί το Cache pollution, περιορίζοντας το μέγεθος της μνήμης cache που μπορούν να καταλάβουν τα prefetched δεδομένα.

Το Cache eviction περιγράφει τις καταστάσεις στις οποίες η συνολική ποσότητα των δεδομένων είναι μεγαλύτερη από το διαθέσιμο ποσό της μνήμης. Όταν η Cache μνήμη είναι πλήρης και ένα νέο στοιχείο προστίθεται, η μνήμη cache πρέπει να εκδιώξει ένα από τα στοιχεία της προκειμένου να δημιουργηθεί χώρος για το νέο στοιχείο. Συνήθως τα παλιά,

σχετικώς άχρηστα, ή υπερβολικά ογκώδης δεδομένα μπορεί να εκδιωχθούν από τη μνήμη cache.

3.7 Inclusive Cache και Exclusive cache

Σε ορισμένους επεξεργαστές, όλα τα δεδομένα στη μνήμη cache L1 πρέπει επίσης να είναι κάπου στη μνήμη L2 cache. Αυτές οι caches ονομάζονται strictly inclusive. Άλλοι επεξεργαστές (όπως ο AMD Athlon) έχουν exclusive caches. Δηλαδή τα δεδομένα - δεδομένων είναι εγγυημένα ότι θα είναι το πολύ σε μία από τις caches L1 και L2 ποτέ και στις δύο. Ο επεξεργαστής που χρησιμοποιούμε για τα πειράματα μας έχει inclusive cache.

Κεφάλαιο 4

Τρόποι-Συναρτήσεις για Software Prefetching

4.1 void __builtin_prefetch 27
4.2 void _mm_prefetch 28
4.3 GCC inline assembly 30

4.1 void __builtin_prefetch (const void *addr , rw, locality)

Αυτή η συνάρτηση χρησιμοποιείται για την ελαχιστοποίηση του cache miss- latency με την μεταφορά των δεδομένων σε μια μνήμη cache πριν να γίνει η πρόσβαση σε αυτά. Μπορούμε να εισάγουμε τις κλήσεις __builtin_prefetch στον κώδικα για τον οποίο γνωρίζουμε τις διευθύνσεις των δεδομένων στη μνήμη που είναι πιθανό να προσπελαστούν σύντομα. Εάν ο στόχος τους τις υποστηρίζει, οι εντολές prefetch θα δημιουργηθούν. Εάν το prefetch γίνει αρκετά νωρίς πριν από την πρόσβαση τότε τα δεδομένα θα είναι στη μνήμη cache μέχρι τη στιγμή που θα γίνει η πρόσβαση.

Η αξία των addr είναι η διεύθυνση της μνήμης που θέλουμε να κάνουμε prefetch . Υπάρχουν δύο προαιρετικά ορίσματα, το rw και το locality. Το rw είναι μια σταθερά κατά την ώρα της μεταγλώτισης, που έχει τιμή ένα ή μηδέν. Αν έχει ένα σημαίνει ότι το prefetch προετοιμάζεται για μια εγγραφή στη διεύθυνση μνήμης και αν έχει μηδεν, που είναι και η προεπιλογή (default), σημαίνει ότι το prefetch προετοιμάζεται για μια ανάγνωση. Το locality είναι μια ακέραια σταθερά κατά την ώρα της μεταγλώτισης, που έχει τιμή μεταξύ μηδέν και τρία. Η τιμή μηδέν σημαίνει ότι τα δεδομένα δεν έχουν καμία χρονική τοπικότητα, έτσι δεν χρειάζεται να μένει στη μνήμη μετά από την πρόσβαση. Η τιμή τρία σημαίνει ότι τα δεδομένα έχουν ένα υψηλό

βαθμό χρονικής τοπικότητας και θα πρέπει να μείνει σε όλα τα πιθανά επίπεδα της μνήμης cache . Οι τιμές ένα και δύο σημαίνουν, αντίστοιχα χαμηλό ή μέτριο βαθμό χρονικής τοπικότητας. Η προεπιλογή είναι η τιμή τρία.

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* … */
  }
```

Το prefetch δεδομένων δεν δημιουργεί σφάλματα, αν η παράμετρος εισόδου addr είναι άκυρη, αλλά η διατύπωση της διεύθυνση πρέπει να είναι έγκυρη. Για παράδειγμα, αν κάνω prefetch p->next δεν θα είναι σφάλμα αν p->next δεν είναι μια έγκυρη διεύθυνση, αλλά θα προκαλέσει σφάλμα η αξιολόγηση σφάλμα αν p δεν είναι μια έγκυρη διεύθυνση.

Εάν ο στόχος δεν υποστηρίζει prefetch δεδομένων, η διατύπωση της διεύθυνση αξιολογείται σωστά εάν περιλαμβάνει παρενέργειες, αλλά κανείς άλλος κώδικας δεν παράγεται και ο GCC δεν εκδίδει οποιαδήποτε προειδοποίηση.

4.2 void _mm_prefetch(char * p , int i);

Η συνάρτηση αυτή φορτώνει μία γραμμή cache δεδομένων από την διεύθυνση p σε μια θέση πιο κοντά στον επεξεργαστή. Η τιμή του i καθορίζει τον τύπο της λειτουργίας prefetch: θα πρέπει να χρησιμοποιούνται οι σταθερές _MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, και _MM_HINT_NTA, που αντιστοιχούν στον τύπο της εντολής prefetch. Επίσης για να την χρησιμοποιήσω πρέπει να συμπεριλάβω στο πρόγραμμα μου και την βιβλιοθήκη #include <emmintrin.h>.

Τα προγράμματα μπορούν να χρησιμοποιήσουν το _mm_prefetch σε κάθε δείκτη στο πρόγραμμα. Οι περισσότεροι επεξεργαστές (σίγουρα όλοι οι x86 και x86-64 επεξεργαστές) αγνοούν τα σφάλματα που προκύπτουν από μη έγκυρους δείκτες που αυτό κάνει πολύ ευκολότερη τη ζωή του προγραμματιστή. Αν ο δείκτης της παραμέτρου εισόδου της συνάρτησης κάνει σε αναφορές έγκυρη μνήμη, θα δοθούν οδηγίες στη μονάδα prefetch για να φορτώσει τα δεδομένα στη μνήμη cache και, εάν είναι απαραίτητο, να εκδιώξει άλλα υπάρχον δεδομένα. Τα περιττά prefetches πρέπει οπωσδήποτε

να αποφεύγονται, γιατί αυτό μπορεί να μειώσει την αποτελεσματικότητα των caches και να καταναλώνει το εύρος ζώνης (bandwidth) της μνήμης (πιθανόν για δύο γραμμές cache σε περίπτωση που η γραμμή της cache που εκδιώχθηκε είναι βρώμικη).

Οι διάφορες τιμές της παραμέτρου εισόδου i που πρέπει να χρησιμοποιούνται με την _mm_prefetch ορίζονται στην υλοποίηση. Αυτό σημαίνει ότι κάθε έκδοση του επεξεργαστή μπορεί να τις εφαρμόσει (ελαφρώς) διαφορετικά. Τι μπορεί γενικά να λεχθεί είναι ότι _MM_HINT_T0 προσκομίζει τα δεδομένα σε όλα τα επίπεδα της μνήμης cache για inclusive caches και στο χαμηλότερο επίπεδο cache για exclusive caches. Εάν το αντικείμενο δεδομένων είναι σε ένα υψηλότερο επίπεδο cache θα φορτωθεί στην L1d. Η τιμή _MM_HINT_T1 τραβά τα δεδομένα στην L2 και όχι στην L1d. Αν υπάρχει κρυφή μνήμη L3 η τιμή _MM_HINT_T2 μπορεί να κάνει κάτι παρόμοιο για αυτήν.

Αυτά είναι λεπτομέρειες, όμως, οι οποίες καθορίζονται ασθενώς και πρέπει να ελεγχθούν για τον πραγματικό επεξεργαστή που χρησιμοποιούμαι. Σε γενικές γραμμές, αν τα δεδομένα πρόκειται να χρησιμοποιηθούν αμέσως το σωστό είναι να χρησιμοποιήσουμε την τιμή _MM_HINT_T0. Φυσικά αυτό προϋποθέτει ότι το μέγεθος της μνήμης L1d cache είναι αρκετά μεγάλο ώστε να χωράει όλα τα δεδομένα που κάναμε prefetch. Εάν το μέγεθος του αμέσως χρησιμοποιημένου σύνολο εργασίας είναι πολύ μεγάλο, το να τα κάνουμε prefetch στην L1d είναι μια κακή ιδέα και οι άλλες δύο τιμές θα πρέπει να χρησιμοποιούνται.

Η τέταρτη τιμή, _MM_HINT_NTA, έχει την ιδιαιτερότητα ότι επιτρέπει στον επεξεργαστή να χειριστεί την prefetched γραμμή ειδικά. Το NTA σημαίνει non-temporal aligned δηλαδή μη-χρονικά ευθυγραμμισμένο. Το πρόγραμμα ειδοποιεί τον επεξεργαστή ότι οι polluting caches με αυτά τα δεδομένα θα πρέπει να αποφεύγονται όσο το δυνατόν περισσότερο διότι τα δεδομένα χρησιμοποιούνται μόνο για ένα σύντομο χρονικό διάστημα. Ο επεξεργαστής μπορεί ως εκ τούτου, κατά την φόρτωση, να αποφύγει την ανάγνωση των δεδομένων στο κατώτερο επίπεδο της μνήμης cache για υλοποιήσεις των inclusive cache. Όταν τα δεδομένα διώχνονται από την L1d δεν χρειάζεται τα δεδομένα να προωθηθούν μέσα στην L2 ή υψηλότερα, αλλά, αντίθετα, μπορούν να εγγραφούν απευθείας στη μνήμη. Μπορεί να υπάρχουν άλλα κόλπα τα οποία οι σχεδιαστές επεξεργαστών μπορούν να χρησιμοποιήσουν, εάν δοθεί αυτή η τιμή.

Ο προγραμματιστής πρέπει να είναι προσεκτικός με αυτή την τιμή: αν το μέγεθος του άμεσου σύνολου εργασίας είναι πολύ μεγάλο και προκαλεί εκδίωξη μιας γραμμής cache που φορτώθηκε με την τιμή NTA, θα συμβεί επαναφόρτωση από τη μνήμη.

4.3 GCC inline assembly: asm volatile("prefetcht0 (%0)\n" : : "r"(&A[i+5]));

Εδώ χρησιμοποιούμε GCC inline assembly.
Στην πιο πάνω εντολή έχουμε τον καταχωρητή %0 και έχουμε σαν παράμετρο εισόδου την διεύθυνση του πίνακα A[i+5]. Όταν το ο περιορισμός «r» ορίζεται, ο gcc μπορεί να κρατήσει τη μεταβλητή σε οποιοδήποτε από τους διαθέσιμους γενικού σκοπού καταχωρητές.
Εάν η assembly δήλωση μας, πρέπει να εκτελεστεί όπου την βάζουμε, (δηλαδή δεν πρέπει να μετακινηθεί από ένα βρόχο ως βελτιστοποίηση), βάζουμε την λέξη-κλειδί " volatile ".
Σαν εντολή της assembly μπορώ να βάλω για Intel επεξεργαστή τις ακόλουθες :

PREFETCHh Instruction Mnemonic	**Actions**
PREFETCHT0	Temporal data—fetch data into all levels of cache hierarchy: • Pentium III processor—1st-level cache or 2nd-level cache • Pentium 4 and Intel Xeon processor—2nd-level cache
PREFETCHT1	Temporal data—fetch data into level 2 cache and higher • Pentium III processor—2nd-level cache • Pentium 4 and Intel Xeon processor—2nd-level cache
PREFETCHT2	Temporal data—fetch data into level 2 cache and higher • Pentium III processor—2nd-level cache • Pentium 4 and Intel Xeon processor—2nd-level cache
PREFETCHNTA	Non-temporal data—fetch data into location close to the processor, minimizing cache pollution • Pentium III processor—1st-level cache • Pentium 4 and Intel Xeon processor—2nd-level cache

Ενώ για Amd επεξεργαστή τις ακόλουθες:

Prefetch type	**Description**
Load	Reads the data into the L1 data cache; the data is later evicted to the L2 cache. The following instructions perform load prefetches: PREFETCH, PREFETCHT0, PREFETCHT1, and PREFETCHT2.
Store	Reads the data into the L1 data cache and marks the data as modified; the data is later evicted to the L2 cache. The PREFETCHW instruction performs a store prefetch.
Nontemporal	The PREFETCHNTA instruction performs a nontemporal prefetch. The data is read into the L1 data cache; to avoid cache pollution, when a PREFETCHNTA misses in the L2 cache and reads from memory, the data is never evicted to the L2 cache. When a PREFETCHNTA hits in the L2 cache, the data is evicted back to the L2 cache.

Κεφάλαιο 5

Υλοποίηση

5.1Τρόπος υλοποίησης 32
5.2 Παράδειγμα Prefetching 33
5.3 Σύστημα μετρήσεων 33
5.4 Τρόπος συλλογής μετρήσεων-αποτελεσμάτων 34

5.1 Τρόπος υλοποίησης

Για την υλοποίηση της διπλωματικής μου εργασίας εφάρμοσα τις συναρτήσεις του software prefetching που επεξηγήθηκαν στο προηγούμενο κεφάλαιο σε πέντε προγράμματα στην προγραμματιστική γλώσσα C:

1. Απλό παράδειγμα που αθροίζει όλα τα στοιχεία ενός πίνακα.
2. Πρόσθεση πινάκων
3. Πολλαπλασιασμός πινάκων
4. Μετάθεση πινάκων
5. Παράλληλο πρόγραμμα με pthreads που αθροίζει όλα τα στοιχεία ενός πίνακα.

Τα προγράμματα αυτά βρίσκονται στο παράρτημα Α.

Σε αυτά τα προγράμματα ο στόχος ήταν να προσκομίσουμε με τις συναρτήσεις του software prefetching τα δεδομένα που θα χρησιμοποιούσαμε λίγους κύκλους μετά ούτως ώστε να είναι έτοιμα στην μνήμη Cache και να μειωθεί ο χρόνος που θα χρειαστεί να τα παραλάβουμε για να τα επεξεργαστούμε.

5.2 Παράδειγμα Prefetching

```
for (i = 0; i < 1000; i++) {
        for (j = 0; j < 1000000; j++) {
                prefetch(&d[j+10][0]);
                for (k = 0; k < 16; k++) {
                        a = a + d[j][k];
                }
        }
}
```

Σε αυτό το παράδειγμα παρατηρούμε ότι επειδή ξέρουμε τι ακριβώς κάνει το προγραμμα μας μπορούμε να στείλουμε στην μνήμη cache τα δεδομένα που θα επεξεργαστεί μελλοντικά. Οπότε και στέλνουμε κάθε φορά την γραμμή του πίνακα που είναι 10 φορές μπροστά από την γραμμή που επεξεργαζόμαστε. Αυτό γίνεται κυρίως διότι υπάρχει κάποια μικρή καθυστέρηση στο να μεταφερθούν τα δεδομένα στην μνήμη cache γι αυτό και φέρνουμε πιο μακρινές γραμμές του πίνακα.

Επίσης με τις συναρτήσεις του software prefetching που είδαμε στο πιο πάνω κεφάλαιο, μπορούν να εφαρμοστούν σε αυτό το παράδειγμα χωρίς αλλαγές. Κυρίως γιατί η γραμμή του πίνακα μας έχει 16 στοιχεία (στην μνήμη φυλάγονται συνεχόμενα αυτά τα στοιχεία) και εμείς κάθε φορά που κάνουμε prefetching γεμίζουμε μια cache line που στην περίπτωση μας έχει μέγεθος 64 bytes. Άρα επειδή ασχολούμαστε με ακαίραιους που είναι 4 bytes ο καθένας πάνε στην μνήμη cache όπως πρέπει όλα τα δεδομένα της γραμμής ενός πίνακα χωρίς να χάνεται κάτι (64 bytes / 4 bytes = 16 στοιχεία).

5.3 Σύστημα μετρήσεων

Ο υπολογιστής που έτρεξα τα πειράματα μου είχε τα ακόλουθα χαρακτηριστικά:
Διπύρηνος επεξεργαστής: Intel(R) Core(TM)2 Duo CPU P8600
Συχνότητα κάθε πυρήνα: 2.40GHz
Μνήμη RAM: 3 GB
Μέγεθος μνήμης cache L1: 32KB
Μέγεθος μνήμης cache L2: 3072KB

Cache line size: 64 Bytes
Associativity μνήμης cache : 8-way set associative
Inclusive cache.

5.4 Τρόπος συλλογής μετρήσεων-αποτελεσμάτων

Από κάθε πρόγραμμα που έτρεξα μέσω μιας συνάρτησης (που μπορείται να δείται στο παράρτημα Α) πήρα τον χρόνο εκτέλεσης του. Επίσης με την εντολή του Linux perf πήρα τα L1 και L2 Cache misses (η εντολή perf στο παράρτημα Α). Κάθε πρόγραμμα το έτρεξα 100 φορές και πήρα τα συνολικά αποτελέσματα και βρήκα το μέσο όρο τους.
Επίσης όλα τα προγράμματα μεταγλωττίστηκαν σε περιβάλον ubuntu με τον gcc compiler και την παράμετρο –O3 (μέγιστο optimization).

Κεφάλαιο 6

Μετρήσεις-Αποτελέσματα

6.1 Μετρήσεις για κάθε πρόγραμμα 35
6.2 Γραφικά Αποτελέσματα 37
6.3 Επιπλέον Μετρήσεις 44
6.4 Επιπλέον Γραφικές Παραστάσεις 46

6.1 Μετρήσεις

Για κάθε πρόγραμμα έκανα 100 εκτελέσεις και πήρα το μέσο όρο τους για να καταλήξω στις ακόλουθες πιο κάτω μετρήσεις.

6.1.1 Απλό παράδειγμα που αθροίζει όλα τα στοιχεία ενός πίνακα.

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
Without prefetching	12.745752	11,326,040,799	32,086,380
__builtin_prefetch()	10.177980	1,169,987,942	270,691,624
_mm_prefetch()	10.171393	1,169,691,566	270,683,644
Asm command	10.180748	1,157,146,022	268,899,702

6.1.2 Πρόσθεση πινάκων

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
Without prefetching	0.00036595	127,192	6,767
__builtin_prefetch()	0.00208537	31,081	7,028
_mm_prefetch()	0.00217781	31,312	7,134
Asm command	0.00221122	31,014	8,406

6.1.3 Πολλαπλασιασμός πινάκων

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
Without prefetching	0.430685	5,470,206	79,058
__builtin_prefetch()	1.432955	1,755,842	115,548
_mm_prefetch()	1.431103	1,738,908	95,190
Asm command	1.423701	1,930,032	122,273

6.1.4 Μετάθεση πινάκων

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
Without prefetching	0.6847231	44,166,991	8,586,350
__builtin_prefetch()	0.5788517	34,521,210	8,760,047
_mm_prefetch()	0.5786672	34,482,052	8,733,126
Asm command	0.5788254	34,628,236	8,793,706

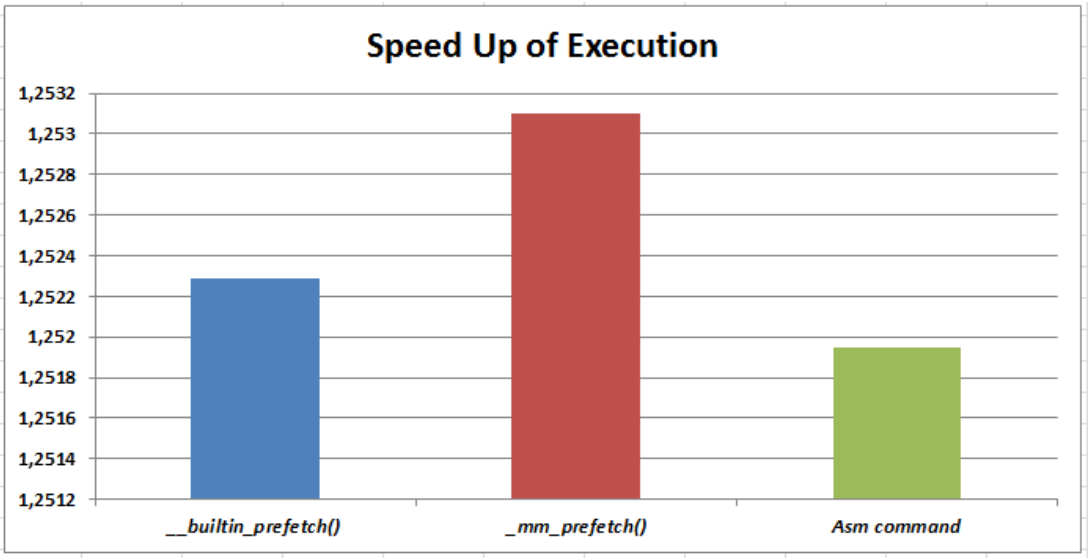
6.1.5 Παράλληλο πρόγραμμα με pthreads που αθροίζει όλα τα στοιχεία ενός πίνακα.

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
Without prefetching	13.06687	11,686,912,686	38,432,570
__builtin_prefetch()	10.83254	1,013,912,612	878,006,022
_mm_prefetch()	10.37593	1,013,972,828	879,636,145
Asm command	10.3838	1,013,866,365	877,382,957

6.2 Γραφικά Αποτελέσματα

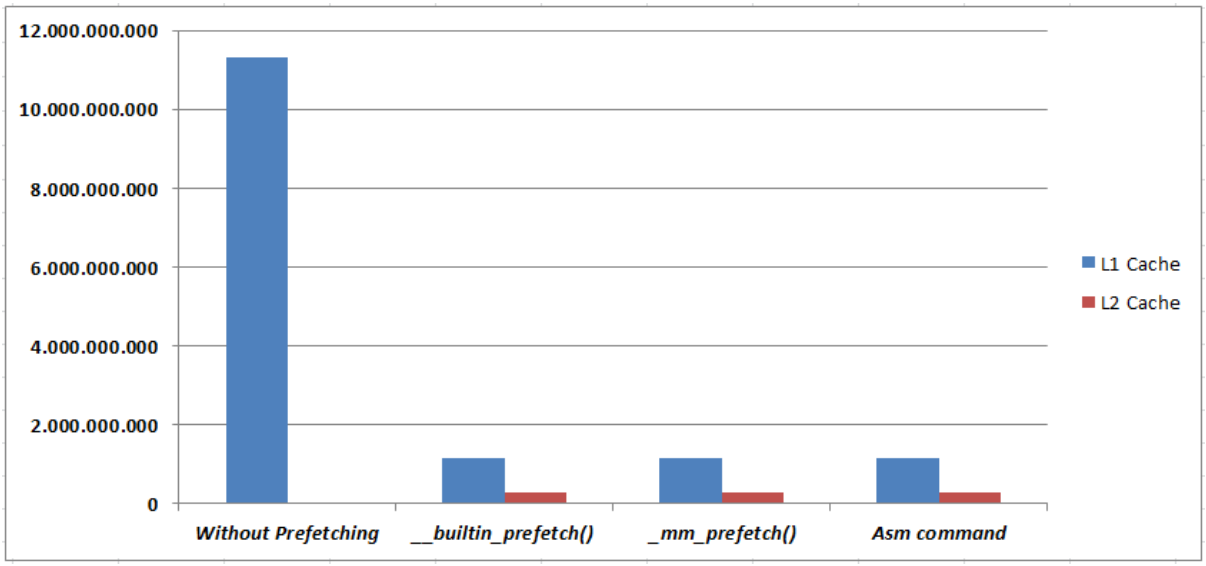
6.2.1 Απλό παράδειγμα που αθροίζει όλα τα στοιχεία ενός πίνακα.

Speed Up εκτέλεσης:



Παρατηρούμε και από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching είναι πιο γρήγορες από την υλοποίηση χωρίς user software prefetching κατά 25%. Αυτό οφείλεται το ότι τα δεδομένα μας ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching και δεν χρειάστηκε να πάει να τα πάρει από άλλου είδους μνήμη που είναι πιο αργή γιατί είναι πιο μακριά από τον πυρήνα μας. Με μικρή διαφορά η πιο γρήγορη υλοποίηση είναι το _mm_prefetch() ακολουθημένο από το __builtin_prefetch() και τελευταία είναι η υλοποίηση με την εντολή assembly.

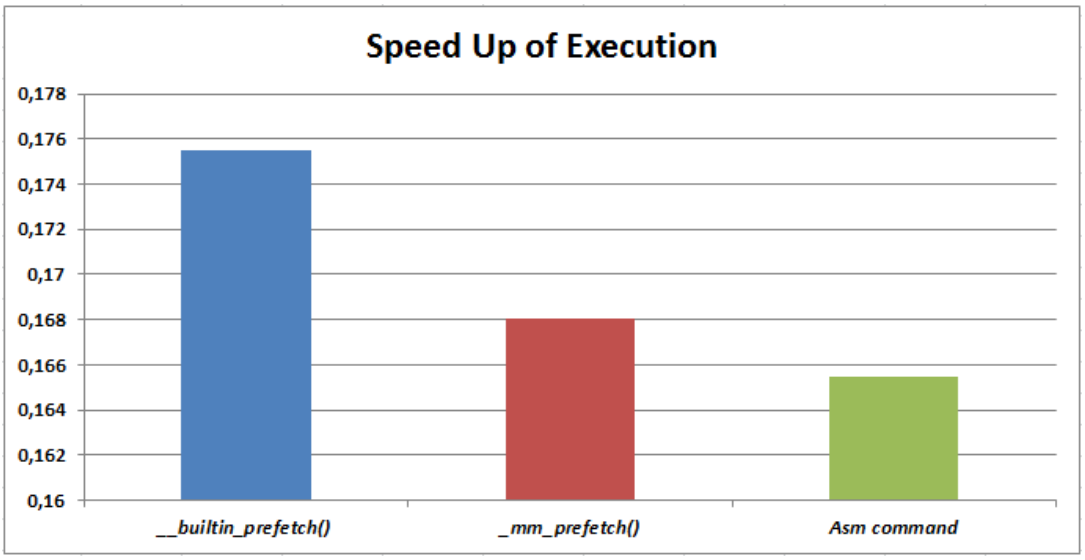
Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching έχουν πολύ πιο λίγα L1 misses από την υλοποίηση χωρίς user software prefetching . Αυτό οφείλεται στο ότι τα δεδομένα ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching . Αντιθέτως όσον αφορά τα L2 misses παρατηρείται ότι τα πιο λίγα με σχετική διαφορά τα έχει η υλοποίηση χωρίς user software prefetching. Αυτό πιθανώς να συμβαίνει γιατί με τις εντολές που χρησιμοποιούμε επηρεάζουμε την μνήμη L2 cache αρνητικά φέρνοντας σε αυτή λάθος δεδομένα την κατάλληλη στιγμή, ή σωστά δεδομένα την λάθος στιγμή. Οι τιμές που παρουσιάζουν όλες οι υλοποιήσεις του software prefetching είναι πολύ κοντά σε όλες τις περιπτώσεις.

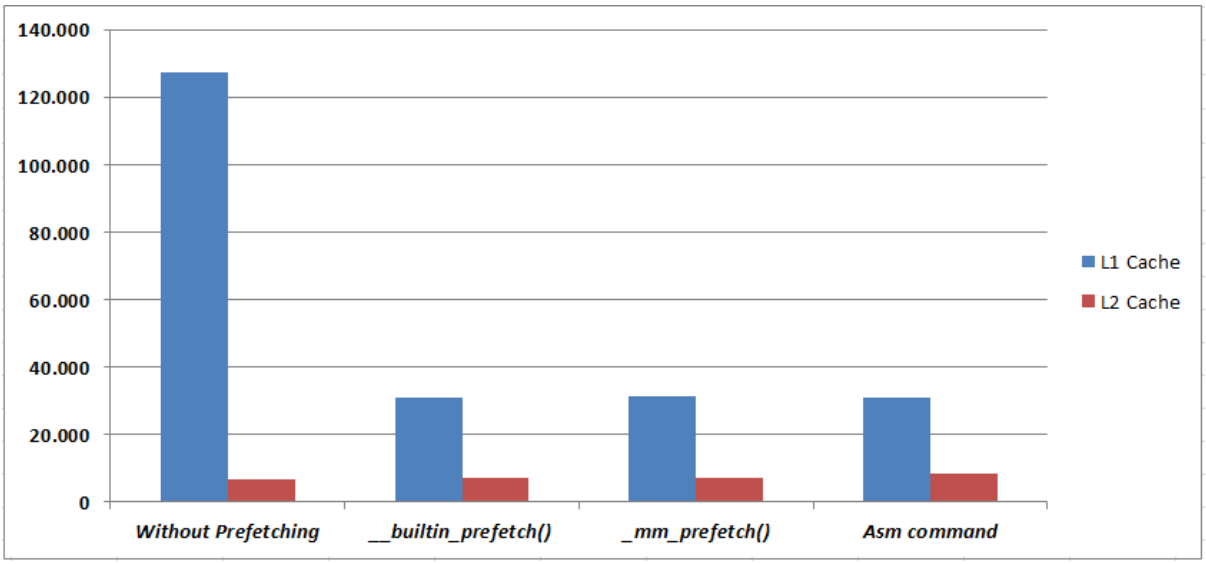
6.2.2 Πρόσθεση πινάκων

Speed Up εκτέλεσης:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching είναι πολύ πιο αργές από την υλοποίηση του χωρίς user software prefetching κατά 80% περίπου. Αυτό μπορεί να οφείλεται στο ότι καλούμε αρκετά συχνά τις συναρτήσεις μας (system calls) για το prefetching και αποφέρει κανένα σημαντικό όφελος. Με μικρή διαφορά η πιο γρήγορη υλοποίηση είναι το __builtin_prefetch() ακολουθημένο από το _mm_prefetch() και τελευταία είναι η υλοποίηση με την εντολή assembly.

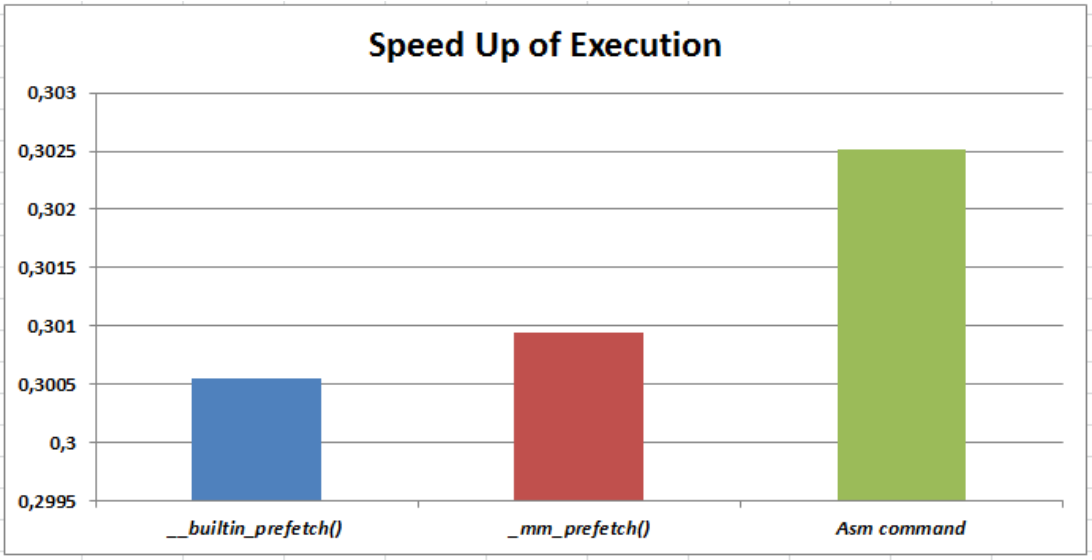
Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching έχουν πολύ πιο λίγα L1 misses από την υλοποίηση χωρίς user software prefetching . Αυτό οφείλεται στο ότι τα δεδομένα ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching. Επίσης όσον αφορά τα L2 misses παρατηρείται ότι τα πιο λίγα με πολύ μικρή διαφορά τα έχει η υλοποίηση χωρίς user software prefetching. Οι τιμές που παρουσιάζουν όλες οι υλοποιήσεις του software prefetching είναι πολύ κοντά σε όλες τις περιπτώσεις.

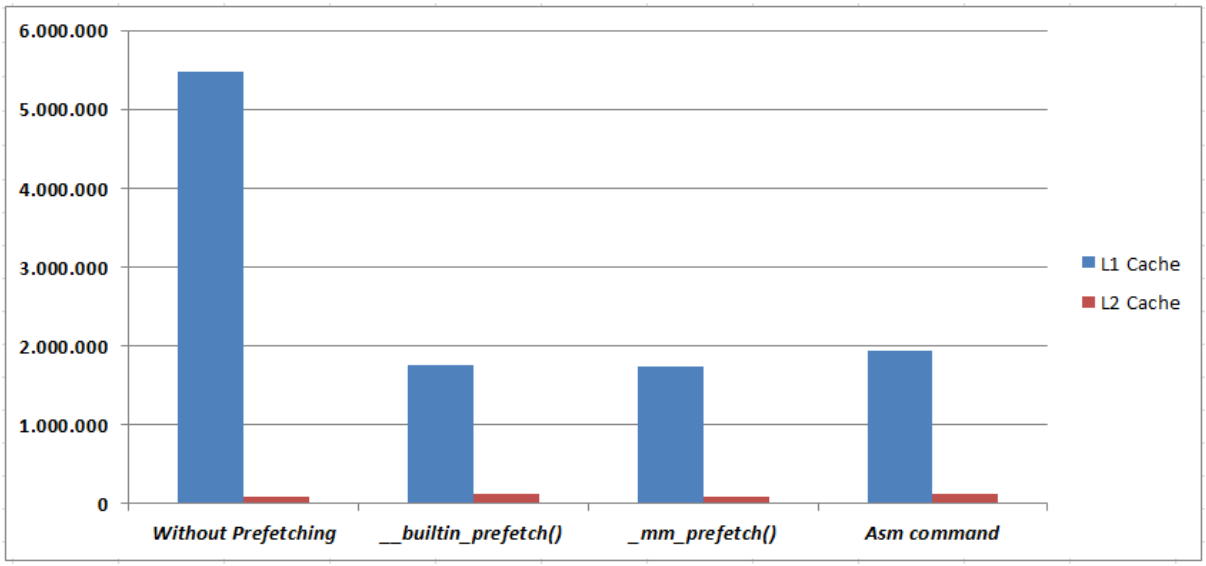
6.2.3 Πολλαπλασιασμός πινάκων

Speed Up εκτέλεσης:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching είναι πολύ πιο αργές από την υλοποίηση του χωρίς user software prefetching κατά 70% περίπου. Αυτό μπορεί να οφείλεται στο ότι καλούμε αρκετά συχνά τις συναρτήσεις μας (system calls) για το prefetching και αποφέρει κανένα σημαντικό όφελος. Με μικρή διαφορά η πιο γρήγορη υλοποίηση είναι της εντολή assembly ακολουθημένο από το _mm_prefetch() και τελευταίο το __builtin_prefetch().

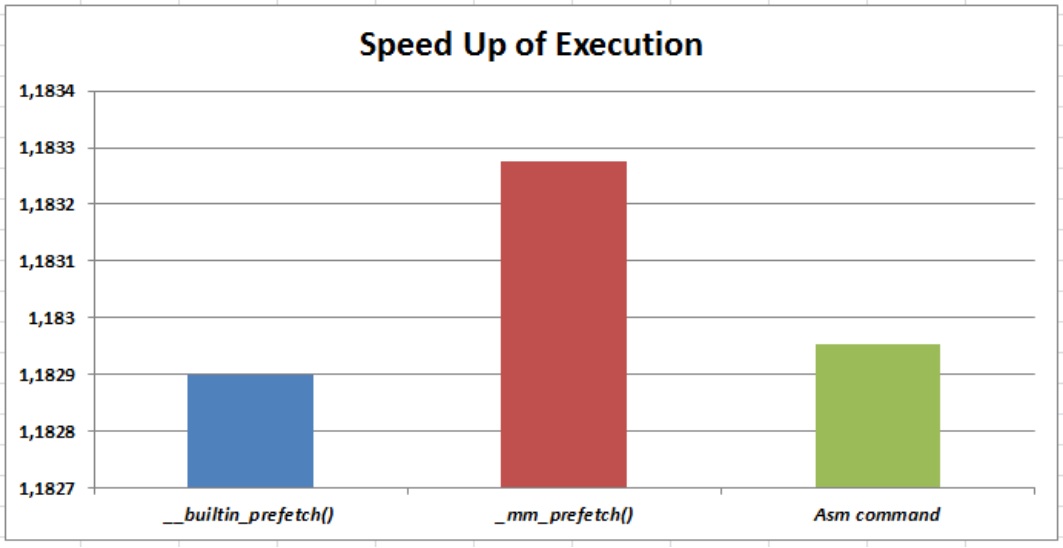
Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching έχουν πολύ πιο λίγα L1 misses από την υλοποίηση χωρίς user software prefetching . Αυτό οφείλεται στο ότι τα δεδομένα ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching .Επίσης όσον αφορά τα L2 misses παρατηρείται ότι τα πιο λίγα με πολύ μικρή διαφορά τα έχει η υλοποίηση χωρίς user software prefetching. Οι τιμές που παρουσιάζουν όλες οι υλοποιήσεις του software prefetching είναι πολύ κοντά με την εντολή assembly να έχει λίγα περισσότερα misses.

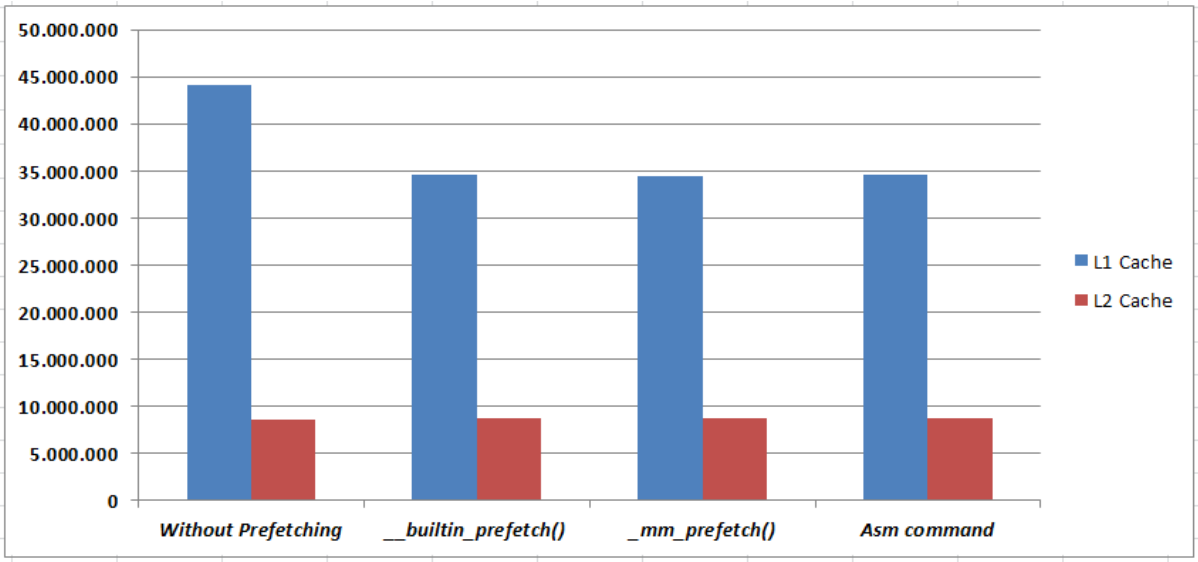
6.2.4 Μετάθεση πινάκων

Speed Up εκτέλεσης:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching είναι αρκετά πιο γρήγορες από την υλοποίηση του χωρίς user software prefetching κατά 18% περίπου. Αυτό οφείλεται το ότι τα δεδομένα μας ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching και δεν χρειάστηκε να πάει να τα πάρει από άλλου είδους μνήμη που είναι πιο αργή γιατί είναι πιο μακριά από τον πυρήνα μας. Με μικρή διαφορά η πιο γρήγορη υλοποίηση είναι το _mm_prefetch() ακολουθημένο από την εντολή assembly και τελευταίο είναι το __builtin_prefetch().

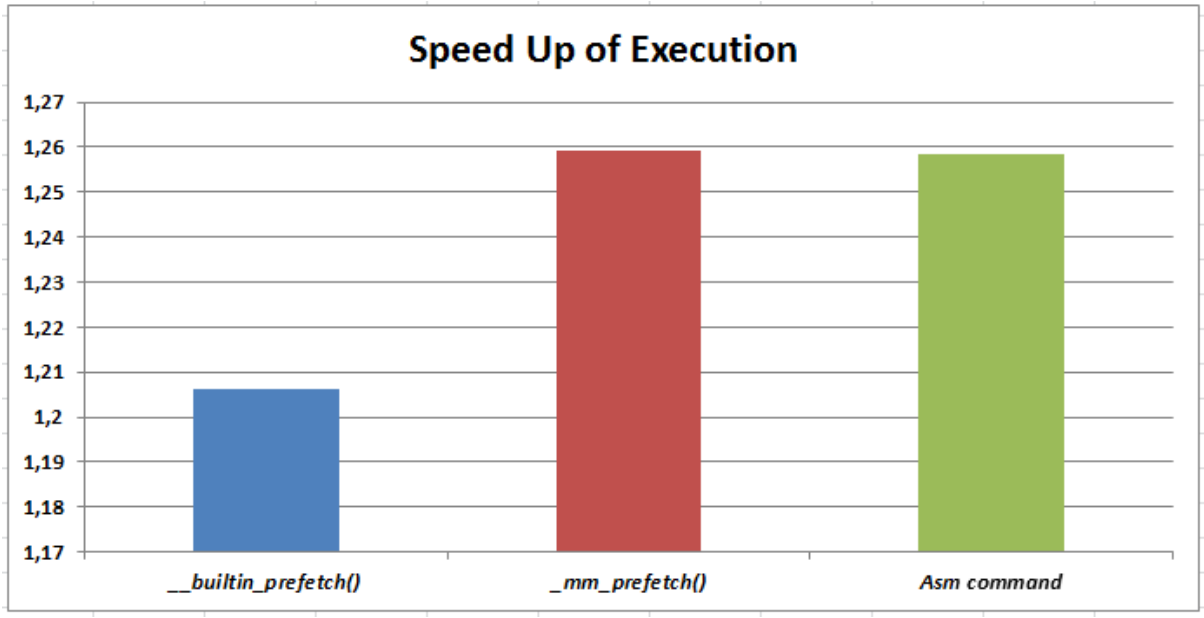
Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching έχουν πολύ πιο λίγα L1 misses από την υλοποίηση χωρίς user software prefetching. Αυτό οφείλεται στο ότι τα δεδομένα ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching . Όσο αφορά τα L2 misses δεν παρατηρείται κάποια σημαντική διαφορά μεταξύ όλων των διαφορετικών υλοποιήσεων. Οι τιμές που παρουσιάζουν όλες οι υλοποιήσεις του software prefetching είναι πολύ κοντά σε όλες τις περιπτώσεις.

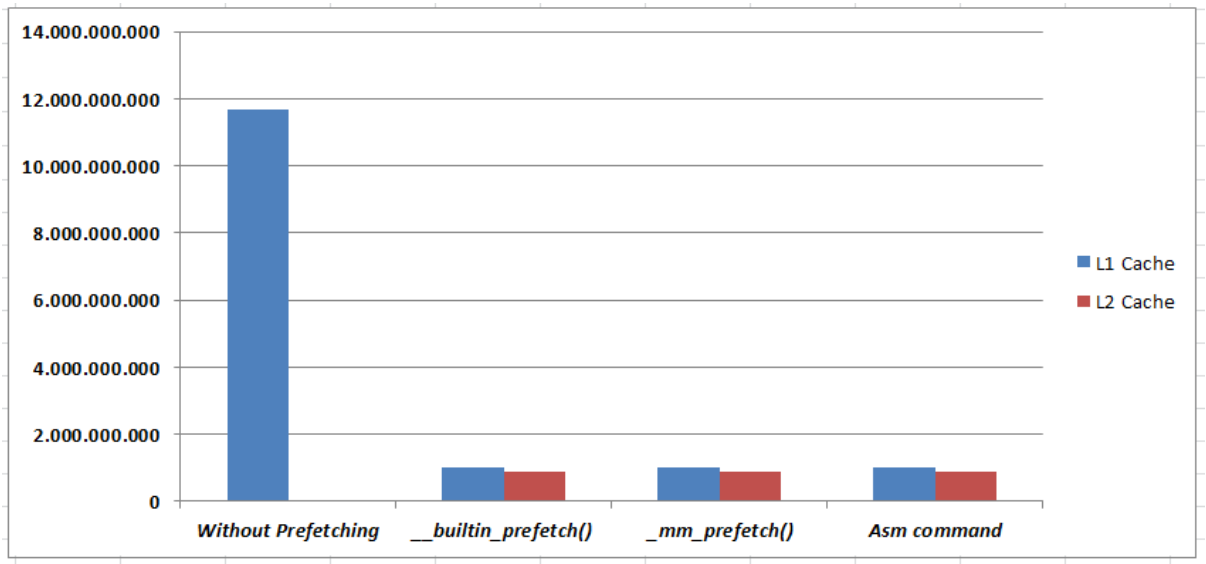
6.2.5 Παράλληλο πρόγραμμα με pthreads που αθροίζει όλα τα στοιχεία ενός πίνακα.

Speed Up εκτέλεσης:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching είναι αρκετά πιο γρήγορες από την υλοποίηση του χωρίς user software prefetching κατά 23% περίπου. Αυτό οφείλεται το ότι τα δεδομένα μας ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching και δεν χρειάστηκε να πάει να τα πάρει από άλλου είδους μνήμη που είναι πιο αργή γιατί είναι πιο μακριά από τον πυρήνα μας. Με μικρή διαφορά η πιο γρήγορη υλοποίηση είναι το _mm_prefetch() ακολουθημένο από την εντολή assembly και τελευταίο το __builtin_prefetch() με κάπως σημαντική διαφορά.

Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι όλες οι υλοποιήσεις του software prefetching έχουν πολύ πιο λίγα L1 misses από την υλοποίηση χωρίς user software prefetching . Αυτό οφείλεται στο ότι τα δεδομένα ήταν στον σωστό χρόνο στην μνήμη cache L1 λόγω της συνάρτησης μας για prefetching . Αντιθέτως όσον αφορά τα L2 misses παρατηρείται ότι τα πιο λίγα με μεγάλη διαφορά τα έχει η υλοποίηση χωρίς user software prefetching. Αυτό πιθανώς να συμβαίνει γιατί με τις εντολές που χρησιμοποιούμε επηρεάζουμε την μνήμη L2 cache αρνητικά φέρνοντας σε αυτή λάθος δεδομένα την κατάλληλη στιγμή, ή σωστά δεδομένα την λάθος στιγμή. Οι τιμές που παρουσιάζουν όλες οι υλοποιήσεις του software prefetching είναι πολύ κοντά σε όλες τις περιπτώσεις.

6.3 Επιπλέον Μετρήσεις

Όλες αυτές οι μετρήσεις έγιναν στο απλό παράδειγμα που αθροίζει όλα τα στοιχεία ενός πίνακα αλλά με διαφορετικές παραμέτρους.

6.3.1 Παίρνοντας διαφορετική διεύθυνση για την συνάρτηση _mm_prefetch.

Address of Prefetch	Time (seconds)	L1 misses	L2 misses
(&d[j+1][0])	12.227890	10,125,922,480	22,538,555
(&d[j+5][0])	10.917750	4,118,776,219	43,928,664
(&d[j+10][0])	10.869860	2,269,718,660	187,852,204
(&d[j+16][0])	10.171393	1,169,691,566	270,683,644
(&d[j+20][0])	10.510370	1,019,859,720	283,501,922

6.3.2 Παίρνοντας διαφορετική παράμετρο για την i που καθορίζει τον τύπο του prefetch για την συνάρτηση _mm_prefetch.

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
MM_HINT_T0	10.171393	1,169,691,566	270,683,644
MM_HINT_T1	11.206767	6,496,288,401	103,435,891
MM_HINT_T2	11.2200418	6,369,282,950	102,809,018
MM_HINT_NTA	10.522109	1,186,863,582	301,991,618

6.3.3 Παίρνοντας διαφορετική παράμετρο που καθορίζει τον τύπο του prefetch για την εντολή της assembly.

Type of Prefetch	Time (seconds)	L1 misses	L2 misses
Prefetcht0	10.180748	1,157,146,022	268,899,702
Prefetcht1	11.0937235	6,414,391,870	97,002,678
Prefetcht2	11.0991894	6,384,660,004	97,352,077
Prefetchnta	10.4264826	1,178,080,033	301,295,103

6.4 Επιπλέον Γραφικές Παραστάσεις

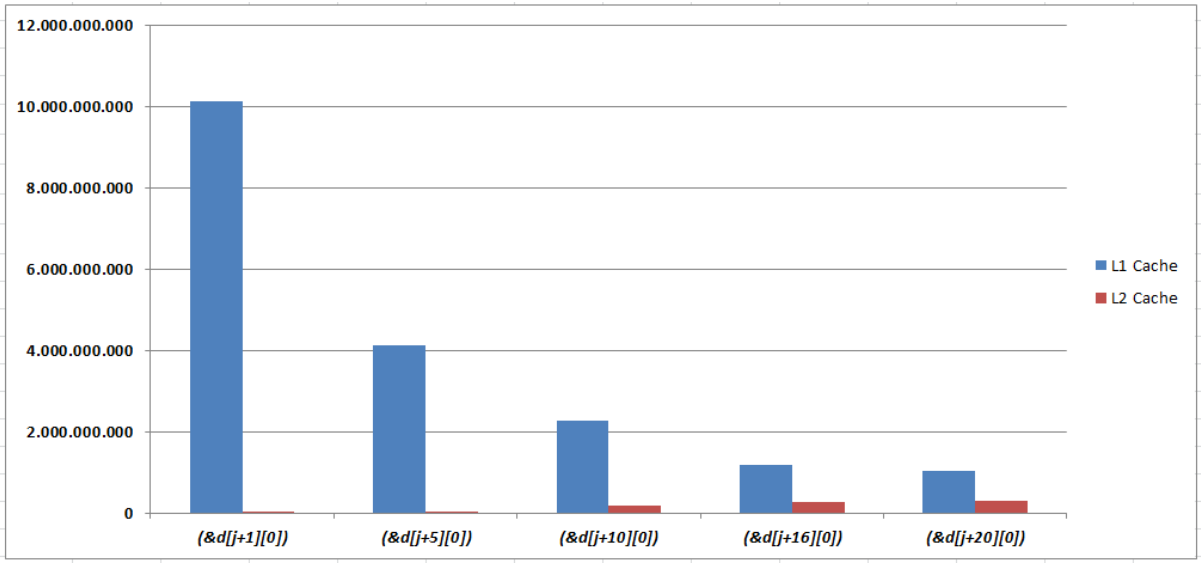
6.4.1 Παίρνοντας διαφορετική διεύθυνση για την συνάρτηση _mm_prefetch.

Speed Up εκτέλεσης:

Παρατηρούμε από την γραφική παράσταση ότι πιο γρήγορη υλοποίηση είναι όταν στέλνουμε κάθε φορά την γραμμή του πίνακα που είναι 16 φορές μπροστά από την γραμμή που επεξεργαζόμαστε. Μετά ακολουθεί η υλοποίηση που στέλνουμε 20 γραμμές μπροστά, μετά η 10 γραμμές , η 5 γραμμές και τέλος αυτή που στέλνουμε την επόμενη γραμμή.

Αυτό γίνεται κυρίως διότι υπάρχει κάποια μικρή καθυστέρηση στο να μεταφερθούν τα δεδομένα στην μνήμη cache γι αυτό και φέρνουμε πιο μακρινές γραμμές του πίνακα και ο χρόνος αυτός είναι πιο κοντά στο όταν φέρνουμε 16 γραμμές μπροστά από την γραμμή που επεξεργαζόμαστε. Όταν φέρνουμε την επόμενη γραμμή όπως είναι λογικό επειδή μέχρι να την φέρει δεν την χρειαζόμαστε οπότε και έχουμε το χειρότερο αποτέλεσμα.

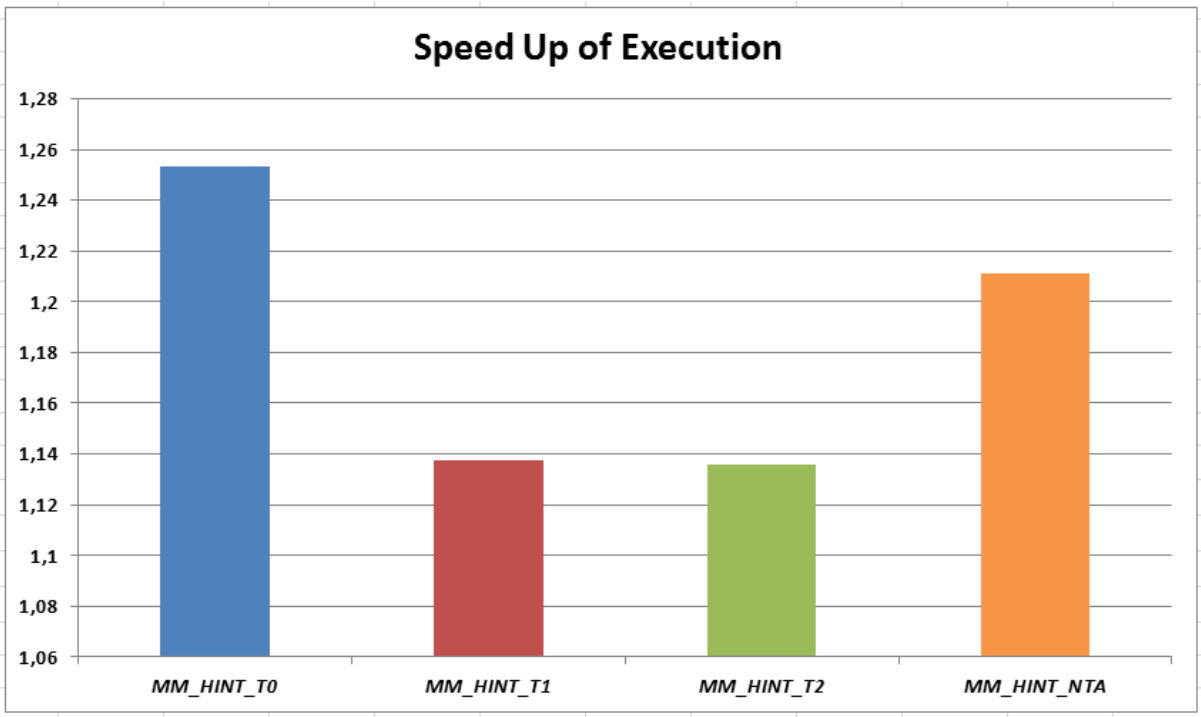
Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι τα πιο λίγα L1 misses τα έχει η υλοποίηση που στέλνουμε 20 γραμμές μπροστά και ακολουθεί με μικρή διαφορά η υλοποίηση γρήγορη που στέλνουμε 16 γραμμές διότι τα δεδομένα που χρειαζόμαστε βρίσκονται συνήθως στην L1 cache οπότε και έχουμε λιγότερα L1 misses. Ενώ τα περισσότερα L1 misses τα έχει η υλοποίηση που στέλνουμε την επόμενη γραμμή επειδή μέχρι να την φέρει στην L1 cache δεν την χρειαζόμαστε οπότε και έχουμε συνεχώς L1 misses. Αντιθέτως όσον αφορά τα L2 misses παρατηρείται ότι τα περισσότερα τα έχουν οι υλοποιήσεις που στέλνουμε 20 γραμμές μπροστά,16 γραμμές και 10 γραμμές με την ακόλουθη σειρά. Αυτό πιθανώς να συμβαίνει γιατί με τις εντολές που χρησιμοποιούμε επηρεάζουμε την μνήμη L2 cache αρνητικά φέρνοντας σε αυτή λάθος δεδομένα την κατάλληλη στιγμή, ή σωστά δεδομένα την λάθος στιγμή παρά στις άλλες δύο περιπτώσεις.

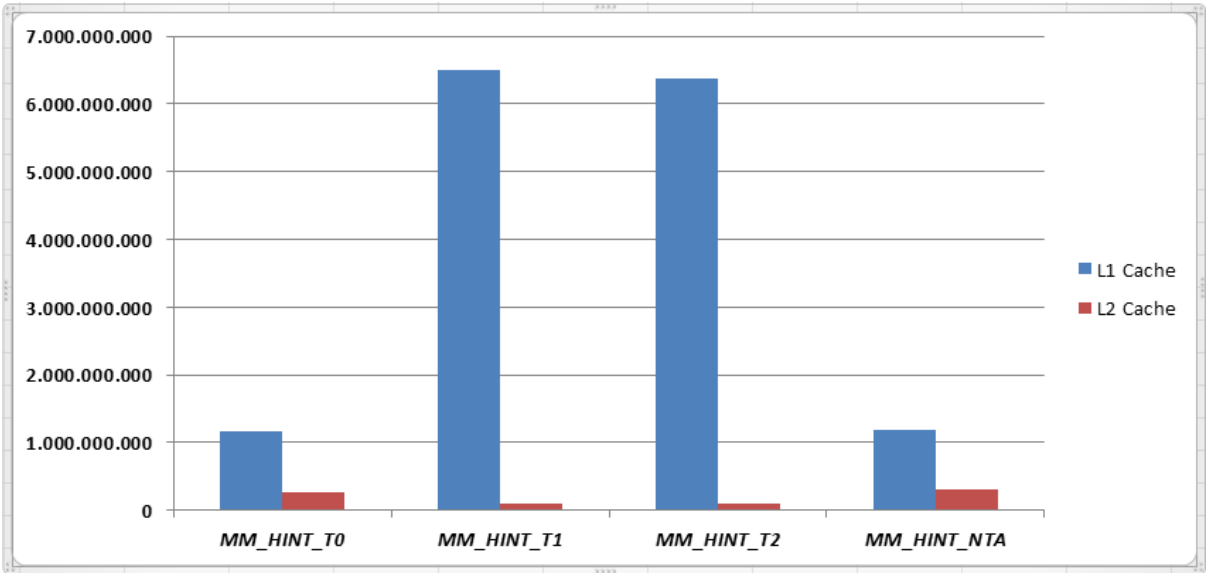
6.4.2 Παίρνοντας διαφορετική παράμετρο για την i που καθορίζει τον τύπο του prefetch για την συνάρτηση _mm_prefetch.

Speed Up εκτέλεσης:



Παρατηρούμε από την γραφική παράσταση ότι πιο γρήγορη υλοποίηση είναι όταν έχουμε MM_HINT_T0 και αυτό λογικά γιατί στέλνουνε τα δεδομένα που χρειαζόμαστε στην L1 cache που είναι πιο κοντά στον επεξεργαστή από την L2 cache οπότε έχουμε και λιγότερη καθυστέρηση να τα φέρουμε. Μετά ακολουθεί η υλοποίηση όταν έχουμε MM_HINT_NTA. Οι πιο αργές υλοποιήσεις είναι οι MM_HINT_T1 και η MM_HINT_T2, που αυτό οφείλεται λογικά στο ότι τα δεδομένα βρίσκονται στην L2 cache που είναι πιο μακριά από τον επεξεργαστή οπότε έχουμε και μεγαλύτερη καθυστέρηση να τα φέρουμε.

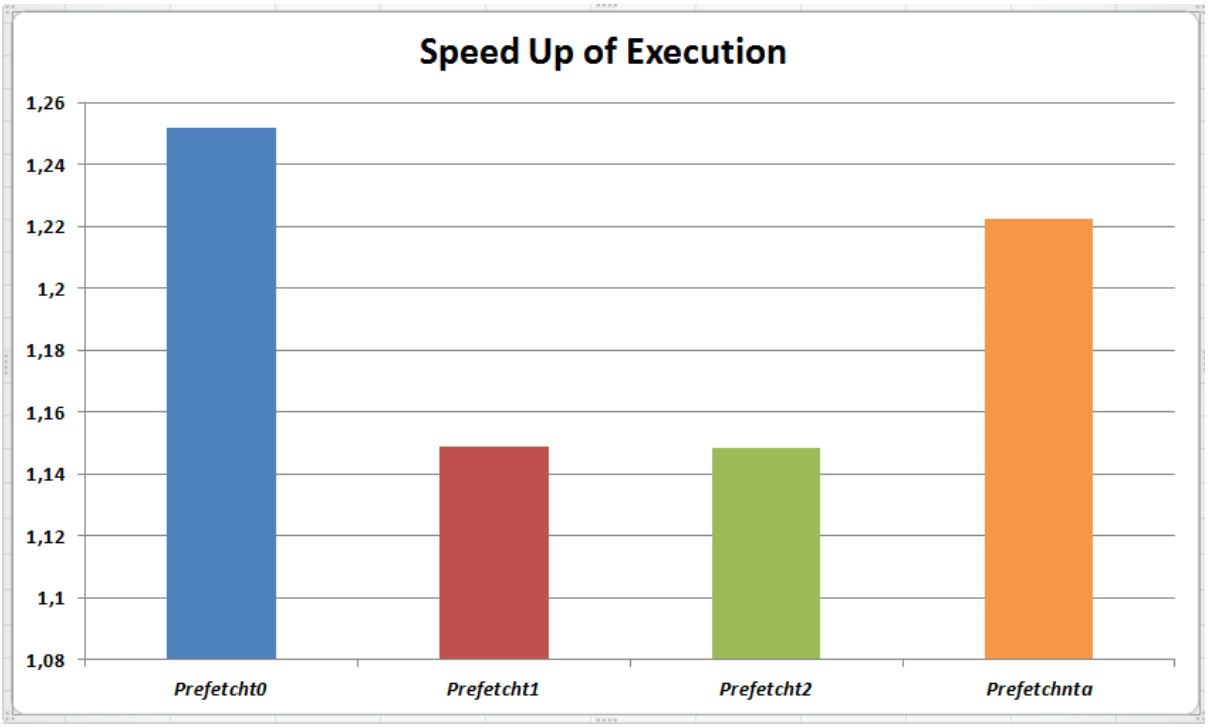
Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι τα πιο λίγα L1 misses τα έχει η υλοποίηση είναι όταν έχουμε MM_HINT_T0 αυτό λογικά γιατί στέλνουνε τα δεδομένα που χρειαζόμαστε στην L1 cache άρα έχουμε λιγότερα L1 misses. Μετά ακολουθεί η υλοποίηση όταν έχουμε MM_HINT_NTA. Τα πιο πολλά L1 misses τα έχουν οι MM_HINT_T1 και η MM_HINT_T2, που αυτό οφείλεται λογικά στο ότι τα δεδομένα βρίσκονται στην L2 cache οπότε και έχουμε περισσότερα L1 misses. Αντιθέτως όσον αφορά τα L2 misses παρατηρείται ότι τα περισσότερα τα έχουν οι MM_HINT_T0 και MM_HINT_NTA γιατί αυτό πιθανώς να συμβαίνει γιατί με τις εντολές που χρησιμοποιούμε επηρεάζουμε την μνήμη L2 cache αρνητικά φέρνοντας σε αυτή λάθος δεδομένα την κατάλληλη στιγμή, ή σωστά δεδομένα την λάθος στιγμή. Ενώ στις άλλες δύο περιπτώσεις φέρνουμε τα δεδομένα στην μνήμη L2 cache οπότε έχουμε και λιγότερα L2 misses.

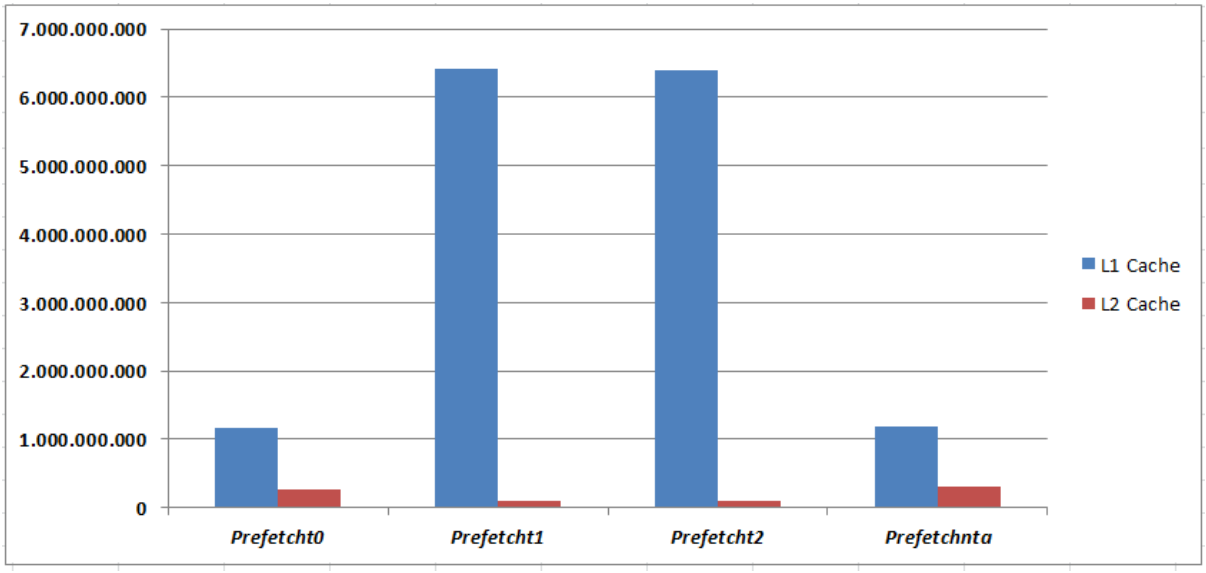
6.4.3 Παίρνοντας διαφορετική παράμετρο που καθορίζει τον τύπο του prefetch για την εντολή της assembly.

Speed Up εκτέλεσης:



Παρατηρούμε από την γραφική παράσταση ότι πιο γρήγορη υλοποίηση είναι όταν έχουμε Prefetcht0 και αυτό λογικά γιατί στέλνουνε τα δεδομένα που χρειαζόμαστε στην L1 cache που είναι πιο κοντά στον επεξεργαστή από την L2 cache οπότε έχουμε και λιγότερη καθυστέρηση να τα φέρουμε. Μετά ακολουθεί η υλοποίηση όταν έχουμε Prefetchnta. Οι πιο αργές υλοποιήσεις είναι οι Prefetcht1 και η Prefetcht2, που αυτό οφείλεται λογικά στο ότι τα δεδομένα βρίσκονται στην L2 cache που είναι πιο μακριά από τον επεξεργαστή οπότε έχουμε και μεγαλύτερη καθυστέρηση να τα φέρουμε.

Cache misses:



Παρατηρούμε από την γραφική παράσταση ότι τα πιο λίγα L1 misses τα έχει η υλοποίηση είναι όταν έχουμε Prefetcht0 αυτό λογικά γιατί στέλνουνε τα δεδομένα που χρειαζόμαστε στην L1 cache άρα έχουμε λιγότερα L1 misses. Μετά ακολουθεί η υλοποίηση όταν έχουμε Prefetchnta. Τα πιο πολλά L1 misses τα έχουν οι Prefetcht1 και η Prefetcht2, που αυτό οφείλεται λογικά στο ότι τα δεδομένα βρίσκονται στην L2 cache οπότε και έχουμε περισσότερα L1 misses. Αντιθέτως όσον αφορά τα L2 misses παρατηρείται ότι τα περισσότερα τα έχουν οι Prefetcht0 και Prefetchnta γιατί αυτό πιθανώς να συμβαίνει γιατί με τις εντολές που χρησιμοποιούμε επηρεάζουμε την μνήμη L2 cache αρνητικά φέρνοντας σε αυτή λάθος δεδομένα την κατάλληλη στιγμή, ή σωστά δεδομένα την λάθος στιγμή. Ενώ στις άλλες δύο περιπτώσεις φέρνουμε τα δεδομένα στην μνήμη L2 cache οπότε έχουμε και λιγότερα L2 misses.

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα 44
7.2 Μελλοντική εργασία 45

7.1 Συμπεράσματα

Το Software Prefetching αποδείχθηκε ότι εάν εφαρμοστεί σωστά και κάτω από τις κατάλληλες προϋποθέσεις μπορεί να φέρει θετικά αποτελέσματα στην εκτέλεση κάποιου προγράμματος μας. Ωστόσο πρέπει να λάβουμε υπόψη μας κάποιους σημαντικούς παράγοντες που θα επηρεάσουν την σωστή εφαρμογή του Software Prefetching.

Για την σωστή υλοποίηση του Software Prefetching πρέπει να γνωρίζουμε κάποια πράγματα για την αρχιτεκτονική της μηχανής μας. Χρειάζεται να ξέρουμε τα διάφορα επίπεδα της cache, τα μεγέθη τους, και το μέγεθος του cache line μας. Ώστε να καταλάβουμε πού να εφαρμόσουμε το Software Prefetching, πότε να το εφαρμόσουμε και πόσα δεδομένα να παίρνουμε κάθε φορά.

Τα δεδομένα που επεξεργαζόμαστε πρέπει να είναι ευθυγραμμισμένα στην μνήμη και να ξέρουμε τι στέλνουμε με το Software Prefetching μας, ώστε να μην έχουμε λάθος δεδομένα στην μνήμη cache και να πάρουμε το επιθυμητό αποτέλεσμα. Διαφορετικά με την λανθασμένη χρησιμοποίηση του μπορούμε να μειώσουμε σημαντικά την απόδοση του προγράμματος μας.

Σε πολλές περιπτώσεις ο compiler του GCC (με παράμετρο -O3) με τις βελτιστοποιήσεις και τους αυτοματοποιησμούς του υπερισχύει όσον αφορά την επίδοση του προγράμματος μας.
Για Δυναμικά ορισμένους πινακες, μεταβλητές, δομές κ.τ.λ ο μεταγλωττιστής δεν είναι τόσο ιδανικός όσο με τα ορισμένα στατικά, γιατί δεν μπορεί να προβλέψει εύκολα την ροή του προγράμματος μας. Οπότε και είναι χρήσιμο πολύ το Software Prefetching.

Ακόμη πολύ χρήσιμα είναι και μερικά εργαλεία όπως το perf του Linux που χρησιμοποίησα στην συγκεκριμένη περίπτωση για να βρώ τα misses της μνήμης cache και είναι ένα εργαλείο που μπορεί να μας βοηθήσει αρκετά για να δούμε ορισμένους παράγοντες που διαδραματίζουν ρόλο στην απόδοση του προγράμματος μας.

7.2 Μελλοντική εργασία

Βλέποντας ότι κάποιος μπορεί να πετύχει σημαντική βελτίωση της απόδοσης ενός προγράμματος θα ήταν ενδιαφέρον να δοκιμαστεί κάποια υλοποίηση Software Prefetching σε άλλου είδους επεξεργαστές και ανάλογα και με τις απαιτήσεις του προγράμματος να βρεθεί η καλύτερη δυνατή λύση και οι καλύτερες δυνατές περιπτώσεις για την απόδοση του προγράμματος.

Με το να γνωρίζουμε έτσι καλύτερα το τι, πού και πότε να χρησιμοποιήσουμε Software Prefetching και τα αποτελέσματα και τις συνέπειες ενός καλού ή κακού Software Prefetching θα μπορεί να ενσωματωθεί μελλοντικά και σε πιο μεγάλα προγράμματα και με καλύτερα αποτελέσματα όσον αφορά την απόδοση τους.

Επίσης να ενσωματωθεί prefetching πάνω στο Thread Scheduling Unit του Data-Driven Multithreading μοντέλου που αποτελεί ένα ερευνητικό Project του πανεπιστημίου μου. Δηλαδή θα πρέπει να μπορεί να εφαρμοστεί το Software Prefetching πάνω στα Dthreads αυτού του Data-Driven Multithreading μοντέλου.

Βιβλιογραφία

[1] C. Kyriacou, P. Evripidou and P. Trancoso. “Data- Driven Multithreading Using Conventional Microprocessors”. *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 1176-1188, Oct. 2006.

[2] Z. Wang, D. Burger, K. McKinley, S. K. Reinhardt, C. Weems. “Guided Region Prefetching: A Cooperative Hardware/Software Approach”.

[3] Todd C. Mowry, Monica S. Lam and Anoop Gupta. “Design and Evaluation of a Compiler Algorithm for Prefetching”. 1992.

[4] S. Arandi and P. Evripidou.“DDM-VMc:The Data-Driven Multithreading Virtual Machine for the Cell Processor”.

[5] J. Fritts. Class Lecture, Topic: “Multi-Level Memory Prefetching for Media and Stream Processing”. Department of Computer Science, Washington University.

[6] S. Vander and W. Lilja.“DDM-VMc: “Data Prefetch Mechanisms”. Department of Electrical and Computer Engineering, University of Minnesota.

[7] Jun Yi Lei and R. Allen Jr. Class Lecture, Topic: “An introduction to and analysis of Hardware and Software based Prefetching.” Dec. 11, 2010.

[8] S. Arandi and P. Evripidou. “Programming Multi core Architectures Using Data-Flow Techniques”.

[9] E. Gornish and A. Veidenbaum. “An Integrated Hardware/Software Data Prefetching Scheme for Shared-Memory Multiprocessors”. *International Journal of Parallel Programming* , Vol . 27, No. 1, 1999.

[10] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou and P. Trancoso. “TFlux: A Portable Platform for Data-Driven Multithreading on Commo-dity Multicore Systems”.

[11] P. Trancoso. Class Lecture, Topic: “Advanced Topics – Data-Driven Multithreading.” Department of Computer Science, University of Cyprus, Nicosia, Mar. 9, 2012.

[12] Philip Koopman. Class Lecture, Topic: “Associativity.” Department of Computer Science, Carnegie Mellon University, September 16, 1998.

[13] AMD. “Software Optimization Guide for AMD Family 10h and 12h Processors”. Publication 40546, Revision: 3.13,February 2011.

[14] Intel. “Intel® 64 and IA-32 Architectures Software Developer’s Manual”. March 2012.

[15] Csaba Andras Moritz. Class Lecture, Topic: “An Introduction to Prefetching.” Department of Computer Science, University of Massachusetts, Nicosia, November, 2007.

[16] Surendra Byna, Yong Chen and Xian-He Sun. “A Taxonomy of Data Prefetching Mechanisms”. *The International Symposium on Parallel Architectures, Algorithms, and Networks*. Department of Computer Science Illinois Institute of Technology, Chicago.

Παράρτημα Α

Εδώ θα βάλω τον κώδικα τον προγραμμάτων που υλοποίησα για τους σκοπούς της διπλωματικής μου εργασίας

1. Απλό παράδειγμα που αθροίζει όλα τα στοιχεία ενός πίνακα.

```
#include <sys/time.h>
#include <emmintrin.h>
#include <stdlib.h>
#include <stdio.h>


double gtod_micro()
{
     struct timeval t;
     gettimeofday(&t,NULL);
     return t.tv_usec  + t.tv_sec * 1000000;
}


int a;
int d[1000000][16];

int main()
{
unsigned i, j, k;
double dStart,dEnd;
double elapsed_time;


for(i=0;i<1000000;i++){
        for(j=0;j<16;j++){
            d[i][j]=(rand()% 30);
        }
}

dStart = gtod_micro();
for (i = 0; i < 1000; i++) {
      for (j = 0; j < 1000000; j++) {
            //ENABLE TYPE OF PREFETCH
            //_mm_prefetch(&d[j+16][0],_MM_HINT_T0);
            //asm volatile("prefetcht0 (%0)\n"  : :
      "r"(&d[j+16][0]));
            //__builtin_prefetch (&d[j+16][0]);
            for (k = 0; k < 16; k++) {
                  a = a + d[j][k];
            }
      }
}
```

```
dEnd = gtod_micro();
elapsed_time = (double)((dEnd-dStart)/1000000.0);
printf("%f\n", elapsed_time);

return 0;
}
```

2. Πρόσθεση πινάκων

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <emmintrin.h>


double gtod_micro()
{
     struct timeval t;
     gettimeofday(&t,NULL);
     return t.tv_usec  + t.tv_sec * 1000000;
}

int main (int argc, char *argv[]){

int i, j, m ,c ,d ,k ,RandomSID,TableSize,mod;

//Error Message if fewer arguments are given
if(argc!=4){
     printf("Error! Example of correct program call:\nMatrix
[TableSize] [RandomSID] [Mod]\n");
     exit(1);
    }

//Convert TableSize number to integer
TableSize=atoi(argv[1]);
//Convert RandomSID number to integer
RandomSID=atoi(argv[2]);
//Convert modulo number to integer
mod=atoi(argv[3]);


float A[TableSize][TableSize]__attribute__((aligned(16)));//
declaring matrices of TableSizexTableSize size

/* FILLING MATRICES WITH RANDOM NUMBERS */
srand (RandomSID);

for(i=0;i<TableSize;i++)
    {
        for(j=0;j<TableSize;j++)
        {
            A[i][j]=(rand()% 30)/10;

        } }
```

```
 double dStart,dEnd;
 double elapsed_time;

 dStart = gtod_micro();

 for(i=0;i<TableSize;i++){

      for(j=0;j<TableSize;j++){
            if (j%mod==0){
                  ////ENABLE TYPE OF PREFETCH
                        //__builtin_prefetch (&A[i][j+16]);
                        //__builtin_prefetch (&A[i][j+32]);
                        //asm volatile("prefetcht0 (%0)\n"  : :
"r"(&A[i][j+16]));
                        //asm volatile("prefetcht0 (%0)\n"  : :
"r"(&A[i][j+32]));
                        //_mm_prefetch(&A[i][j+16],_MM_HINT_T0);
                        //_mm_prefetch(&A[i][j+32],_MM_HINT_T0);


                  }
            A[i][j]=A[i][j] + A[i][j];
      }
  }

  dEnd = gtod_micro();
  elapsed_time = (double)((dEnd-dStart)/1000000.0);
  printf("%f\n",elapsed_time);

  return 0;

 }
```

3. Πολλαπλασιασμός πινάκων

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <emmintrin.h>


double gtod_micro()
{
     struct timeval t;
     gettimeofday(&t,NULL);
     return t.tv_usec  + t.tv_sec * 1000000;
}

int main (int argc, char *argv[]){

    int i, j, m ,c ,d ,k ,RandomSID,TableSize,mod;
```

```
//Error Message if fewer arguments are given
if(argc!=4){
     printf("Error! Example of correct program call:\nMatrix
[TableSize] [RandomSID] [Mod]\n");
     exit(1);
}

//Convert TableSize number to integer
TableSize=atoi(argv[1]);
//Convert RandomSID number to integer
RandomSID=atoi(argv[2]);
//Convert modulo number to integer
mod=atoi(argv[3]);


float A[TableSize][TableSize]__attribute__((aligned(16))),
B[TableSize][TableSize],
C[TableSize][TableSize],transpose[TableSize][TableSize]; //
declaring matrices of TableSizexTableSize size


    /* FILLING MATRICES WITH RANDOM NUMBERS */
    srand (RandomSID);

    for(i=0;i<TableSize;i++)
    {
        for(j=0;j<TableSize;j++)
        {
            A[i][j]=(rand()% 30)/10;
            B[i][j]=(rand()% 30)/10;

        }
    }

//Transpose array B
    for( c = 0 ; c < TableSize ; c++ )
    {
        for( d = 0 ; d < TableSize ; d++ )
        {
               transpose[d][c] = B[c][d];
        }
    }

    double dStart,dEnd;
    double elapsed_time;

    dStart = gtod_micro();

    for(i=0;i<TableSize;i++)
    {
        for(j=0;j<TableSize;j++)
        {
           C[i][j]=0; // set initial value of resulting matrix C = 0
```

```
            for(m=0;m<TableSize;m++)
            {
           if (m%mod==0){
                //ENABLE TYPE PREFETCH
                     //__builtin_prefetch (&A[i][m+16]);
                     //__builtin_prefetch (&A[i][m+32]);
                     //asm volatile("prefetcht0 (%0)\n"  : :
"r"(&A[i][m+16]));
                     //asm volatile("prefetcht0 (%0)\n"  : :
"r"(&A[i][m+32]));
                     _mm_prefetch(&A[i][m+16],_MM_HINT_T0);
                     _mm_prefetch(&A[i][m+32],_MM_HINT_T0);
                     }

           C[i][j]=A[i][m]*transpose[j][m]+C[i][j];
            }

        }
    }
 dEnd = gtod_micro();
 elapsed_time = (double)((dEnd-dStart)/1000000.0);
 printf("%f\n", elapsed_time);

 return 0;
}
```

4. Μετάθεση πινάκων

```
#include <stdio.h>
#include <stdlib.h>
#include <emmintrin.h>
#include <string.h>
#include <sys/time.h>


double gtod_micro()
{
     struct timeval t;
     gettimeofday(&t,NULL);
     return t.tv_usec  + t.tv_sec * 1000000;
}

void swap_block(int *A,int *B,int L){

    int *stop = A + L;

    do{
        int tmpA = *A;
        int tmpB = *B;
        *A++ = tmpB;
        *B++ = tmpA;
    }while (A < stop);
}
```

```
void transpose_even(int *T,int block,int x){
    //  Transposes T.
    //  T contains x columns and x rows.
    //  Each unit is of size (block * sizeof(int)) bytes.


    int row_size = block * x;
    int iter_size = row_size + block;

    //  End of entire matrix.
    int *stop_T = T + row_size * x;
    int *end = stop_T - row_size;

    //  Iterate each row.
    int *y_iter = T;
    do{
        //  Iterate each column.
        int *ptr_x = y_iter + block;
        int *ptr_y = y_iter + row_size;

        do{

           //ENABLE TYPE PREFETCH
        //_mm_prefetch((char*)(ptr_y + row_size),_MM_HINT_T0);
      //__builtin_prefetch ((char*)(ptr_y + row_size));
      //asm volatile("prefetcht0 (%0)\n"  : : "r"((char*)(ptr_y +
row_size)));


            swap_block(ptr_x,ptr_y,block);

            ptr_x += block;
            ptr_y += row_size;
        }while (ptr_y < stop_T);

        y_iter += iter_size;
    }while (y_iter < end);
}



int main(){
    int dimension = 3072;
    int block = 16;

    int words = block * dimension * dimension;
    int bytes = words * sizeof(int);


   int *T = (int*)_mm_malloc(bytes,16);


    if (T == NULL){
        printf("Memory Allocation Failure\n");
        exit(1);
    }
    memset(T,0,bytes);
```

```
    double dStart,dEnd;
    double elapsed_time;

    dStart = gtod_micro();

    transpose_even(T,block,dimension);

    dEnd = gtod_micro();
    elapsed_time = (double)((dEnd-dStart)/1000000.0);
    printf("%f\n",elapsed_time);

    _mm_free(T);

    return 0;
}
```

5. Παράλληλο πρόγραμμα με pthreads που αθροίζει όλα τα στοιχεία ενός πίνακα.

```
#include <sys/time.h>
#include <emmintrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#define NUM_OF_THREADS 2

double gtod_micro()
{
     struct timeval t;
     gettimeofday(&t,NULL);
     return t.tv_usec  + t.tv_sec * 1000000;
}


typedef struct{
int i_lower;
int i_higher;
int sum;
}ARGS;

ARGS args[NUM_OF_THREADS]; //NUM_OF_THREADS
int a;
int d[1000000][16];//__attribute__((aligned(16)));

void* f(void* v){
ARGS* arg=(ARGS*)v;
int i,j,k;



for (j = arg->i_lower; j < arg->i_higher; j++) {
//ENABLE TYPE PREFETCH
//_mm_prefetch(&d[j+16][0],_MM_HINT_T0);
//asm volatile("prefetcht0 (%0)\n"  : : "r"(&d[j+16][0]));
//__builtin_prefetch (&d[j+16][0]);
```

```
      for (k = 0; k < 16; k++) {
            arg->sum = arg->sum + d[j][k];
      }
}

return NULL;
}



int main(){

unsigned i, j, k;
double dStart,dEnd;
double elapsed_time;
void *ptr;
int part;
pthread_t tid[NUM_OF_THREADS];

part=1000000/NUM_OF_THREADS;



    for(i=0;i<1000000;i++)
    {
        for(j=0;j<16;j++)
        {
            d[i][j]=(rand()% 30);
        }
    }

  dStart = gtod_micro();
  for (i = 0; i < 1000; i++) {

      for (j=0; j<NUM_OF_THREADS; j++){
            args[j].sum=0;
            args[j].i_lower=j*part;
            args[j].i_higher=(j+1)*(part);
            pthread_create(&tid[j],NULL,f,&args[j]);
      }
      for (j=0; j<NUM_OF_THREADS; j++){
            pthread_join(tid[j],&ptr);
            a= a + args[j].sum;
      }

  }
    dEnd = gtod_micro();
    elapsed_time = (double)((dEnd-dStart)/1000000.0);
    printf("%f\n", elapsed_time);

    return 0;
}
```

Παράρτημα Β

Για να βρω στοιχεία για τον υπολογιστή μου χρησιμοποίησα την εντολή στο command line των Ubuntu**: cat /proc/cpuinfo**

Για να μάθω το μέγεθος του cache line για την μνήμη L1 cache χρησιμοποίησα την ακόλουθη εντολή: **getconf LEVEL1_DCACHE_LINESIZE**

Για να μάθω διάφορες πληροφορίες για την μνήμη cache χρησιμοποίησα τις ακόλουθες εντολές: **sudo dmidecode –t cache**

Dmesg | grep cache

Με το εργαλείο perf χρησιμοποίησα την ακόλουθη εντολή για να βρω τα cache misses:
perf stat -e cache-misses,l1d-loads-misses ./prefetch

Επίσης έτρεχα το πρόγραμμα μου 100 φορές με την ακόλουθη εντολή:
for i in $(seq 1 1 30); do echo "Iteration $i"; ./prefetch ; done