ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Άσκηση 4 - Προηγμένα Θέματα Αρχιτεκτονικής Υπολογιστών

Γρηγόριος Θανάσουλας

gregthanasoulas@gmail.com

A.M: 03114131

23 Ιουνίου 2020

Περιεχόμενα

1 Σκοπός **2**

2 Μέρος Α': Πειραματική Αξιολόγηση **2**

2.1 Ερώτημα 3-1-1 2
2.2 Ερώτημα 3-1-2 4
2.3 Ερώτημα 3-1-3 5
2.4 Ερώτημα 3-1-4 9
2.5 Ερώτημα 3-2 12

3 Μέρος Β': Εκτέλεση Αλγορίθμου Tomasulo **15**

1 Σκοπός

Η άσκηση αυτή αποσκοπεί στη μελέτη των μηχανισμών συγχρονισμού και των πρωτοκόλλων συνάφειας κρυφής μνήμης (cache coherence protocols) σε σύγχρονες πολυπύρηνες αρχιτεκτονικές. Για την αξιολόγηση των μηχανισμών, χρησιμοποιείται το εργαλείο Sniper ώστε να προσομοιωθεί ένα πολυπύρηνο σύστημα με διαφορετικούς μηχανικούς συγχρονισμού.

2 Μέρος Α': Πειραματική Αξιολόγηση

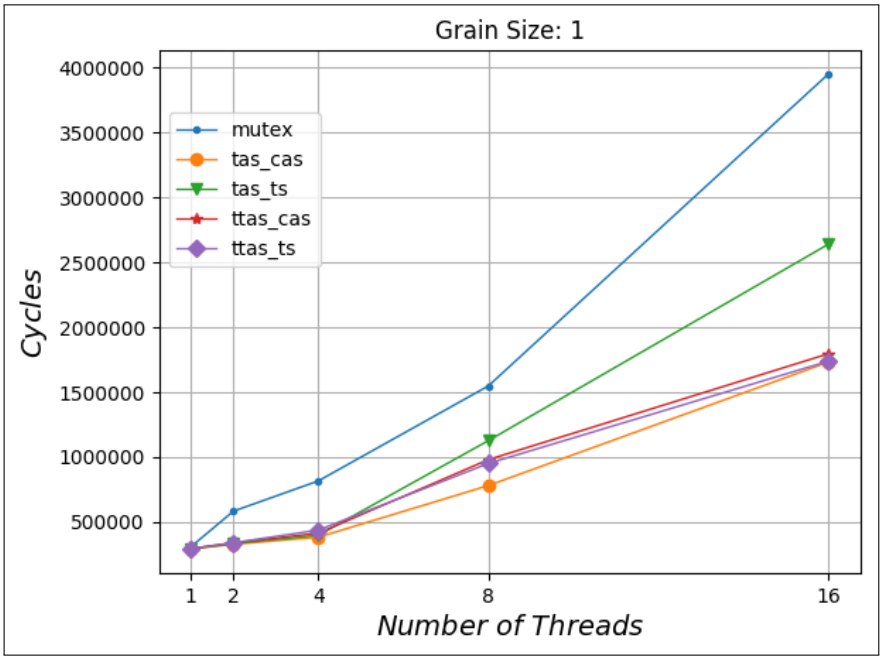
2.1 Ερώτημα 3-1-1

Στο τμήμα αυτό της άσκησης εκτελέστηκαν οι προσομοιώσεις του προγράμματος για όλους τους ακόλουθους συνδυασμούς:

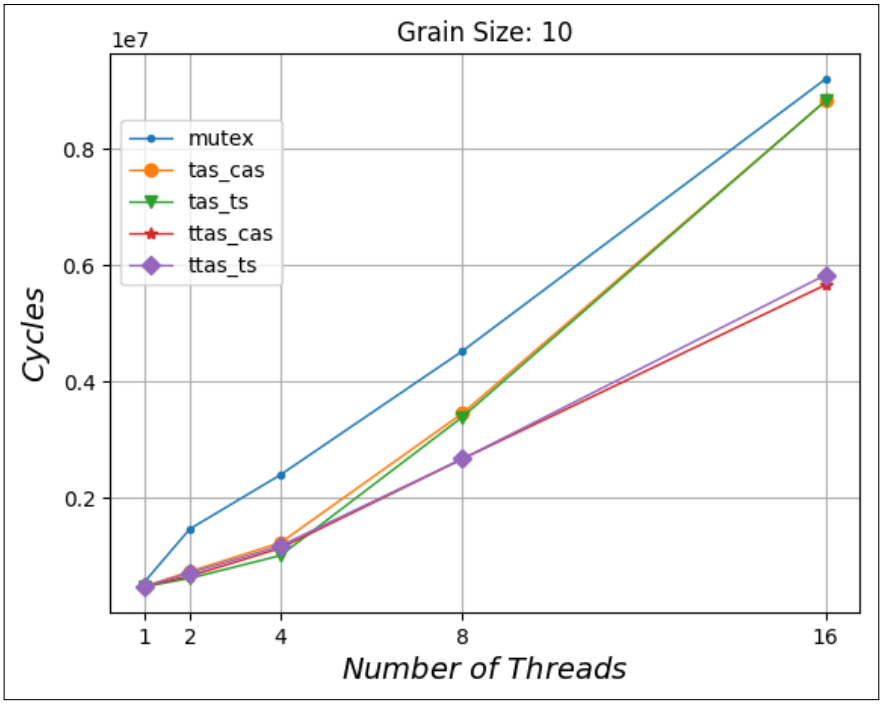
- εκδόσεις προγράμματος:
  - TAS_CAS
  - TAS_TS
  - TTAS_CAS
  - TTAS_TS
  - MUTEX
- iterations: 1000
- nthreads: 1, 2, 4, 8, 16 (σε σύστημα με ισάριθμους πυρήνες)
- grain_size: 1, 10, 100

Ακολουθούν τα σχετικά διαγράμματα που προέκυψαν με βάση τα αρχεία sim.out για τις διαφορετικε παραμέτρους προσομοίωσης:

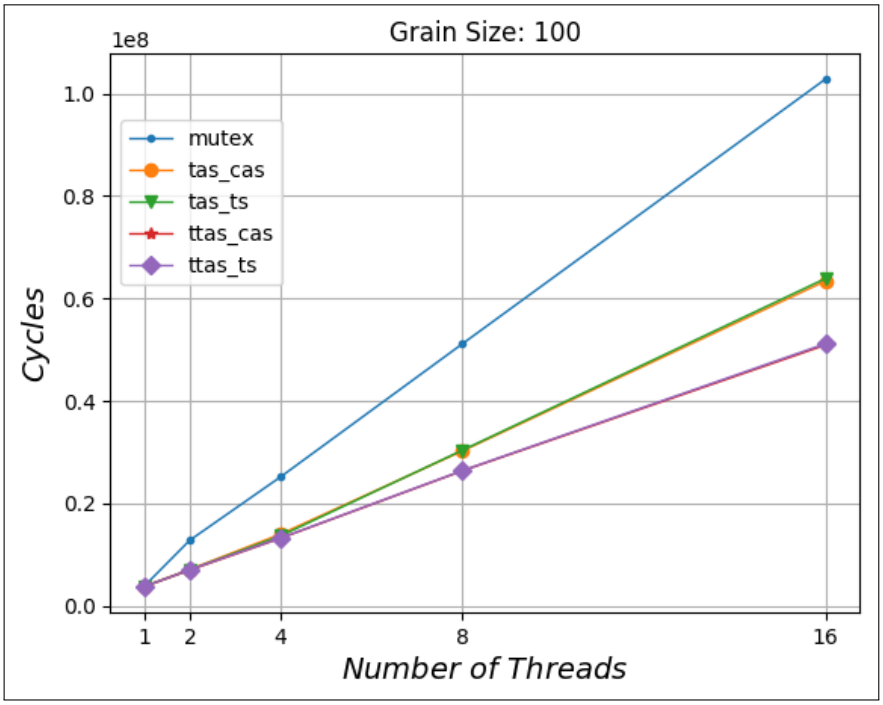
Grain Size: 1



Grain Size: 10



Grain Size: 100



2.2 Ερώτημα 3-1-2

Σχόλια - Συμπεράσματα Οι μηχανισμοί TAS_CAS και TAS_TS υλοποιούν, αμφότεροι το Test-and-set πρωτόκολλο, απλώς χρησιμοποιούν διαφορετικό τρόπο υλοποίησης. Επομένως παρουσιάζουν παραπλήσιες επιδόσεις. Όταν 2 ή περισσότεροι επεξεργαστές προσπαθούν να πάρουν το lock οδηγούνται σε κυκλικές εναλλαγές καταστάσεων (Modified – Invalid) της cache line που περιέχει το lock, δημιουργώντας αυξημένη περιττή κίνηση πάνω στο bus. Οι μηχανισμοί TTAS_CAS και TTAS_TS υλοποιούν το πρωτόκολλο Test-and-Test-and-set. Η διαφορά τους με τις Test-and-set υλοποιήσεις είναι ότι δεν γράφουν συνεχώς πάνω στο lock. Εκτελούν πρώτα ένα load για να δουν αν είναι ελεύθερο και μόνο τότε δοκιμάζουν το atomic instruction που το γράφει και προσπαθεί να το δεσμεύσει δηλαδή πρώτα τσεκάρουν το lock και μετά κάνουν test-and-set. Έτσι σε περίπτωση μη διαθέσιμου lock οι επεξεργαστές κάνουν spinning τοπικά στην cache τους αποφεύγοντας το άχρηστο broadcasting στον διάδρομο. Γι αυτό αναμένουμε οι μηχανισμοί Test-and-Test-and-set να πετυχαίνουν καλύτερη κλιμακωσιμότητα και επίδοση σε

σχέση με τους υπόλοιπους.

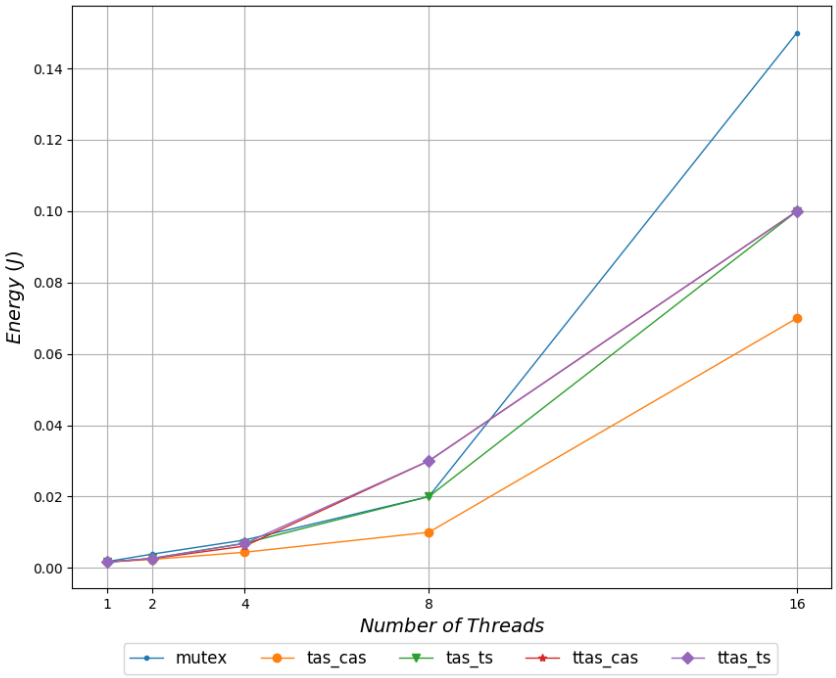
Μελετώντας τα ανωτέρω διαγράμματα, παρατηρούμε πως ο χρόνος της εκτέλεσης (σε κύκλος) αυξάνεται σχεδόν γραμμικά με το πλήθος των νημάτων. Οι ευθείες ωστόσο έχουν διαφορετική κλίση για κάθε διαφορετικό μηχανισμό συγχρονισμού. Για grain size 1 και 100 την μεγαλύτερη κλίση (άρα και τη χειρότερη κλιμακωσιμότητα) εχει ο μηχανισμός MUTEX, που σημαίνει ότι αυξάνει πολύ πιο γρήγορα ο χρόνος εκτέλεσης σε σχέση με τους άλλους μηχανισμούς. Σε όλα τα grain sizes ωστόσο για το εύρος thread 1 έως 16, παρατηρούμε πως οι μηχανισμοί σε σειρά βελτιούμενης κλιμακωσιμότητας και επίδοσης από άποψη χρόνου εκτέλεσης οι TAS_TS & TAS_CAS και τέλος τα TTAS_TS & TTAS_CAS. Από όλα τα διαγράμματα γίνεται αντιληπτό ότι την βέλτιστη κλιμακωσιμότητα -όπως αναμέναμε- σε όλες τις περιπτώσεις επιτυγχάνει ο μηχανισμός Test-and-Test-and-SET (TTAS) υλοποιημένος είτε με τις ατομικές εντολές test-and-set ή compare-and-swap.

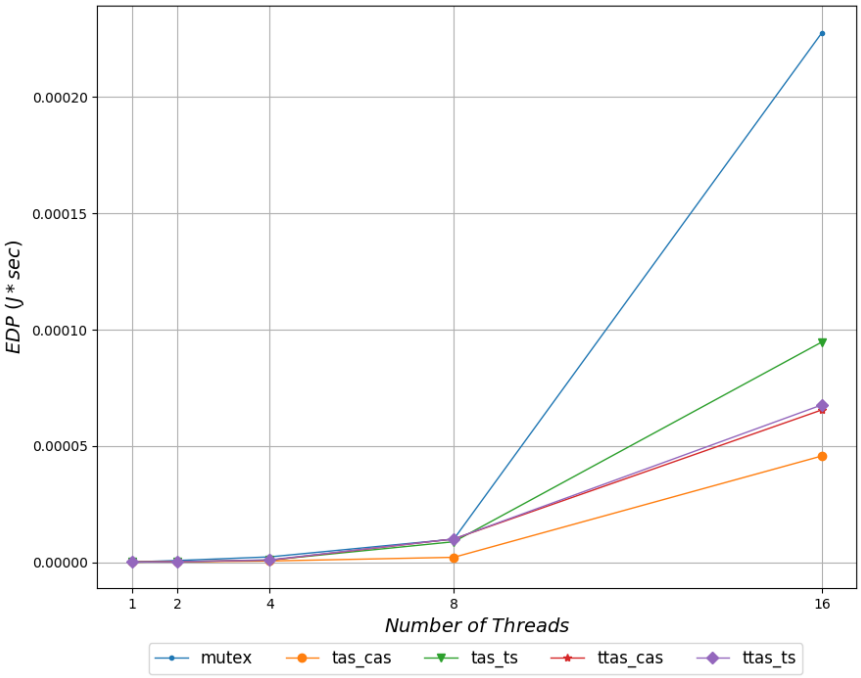
Σχετικά με το grain size παρατηρείται ότι για grain size 10, οι μηχανισμοί TAS για 16 threads πλησιάζουν στην επίδοση τον μηχανισμό MUTEX. Παρατηρούμε επίσης πως καθώς αυξάνεται το grain size, δηλαδή το πλήθος των dummy υπολογισμών, τόσο περισσότερο βελτιώνεται και γίνεται γραμμική με χαμηλότερη κλίση η κλιμακωσιμότητα των μηχανισμών, και ιδίως για το MUTEX. Αυτό μπορεί να αιτιολογηθεί, καθώς σπαταλάται πλέον περισσότερος χρόνος για τους υπολογισμούς και λιγότερος χρόνος για τον ανταγωνισμό του lock, οπότε έχουμε καλύτερη επίδοση. Σαφώς βέβαια, αφού αυξάνει το πλήθος των dummy υπολογισμών οι χρόνοι στα αντίστοιχα διαγράμματα μεγαλώνουν κατά μία τάξη μεγέθους (x10) για κάθε δεκαπλασιασμό του grain size.

2.3 Ερώτημα 3-1-3

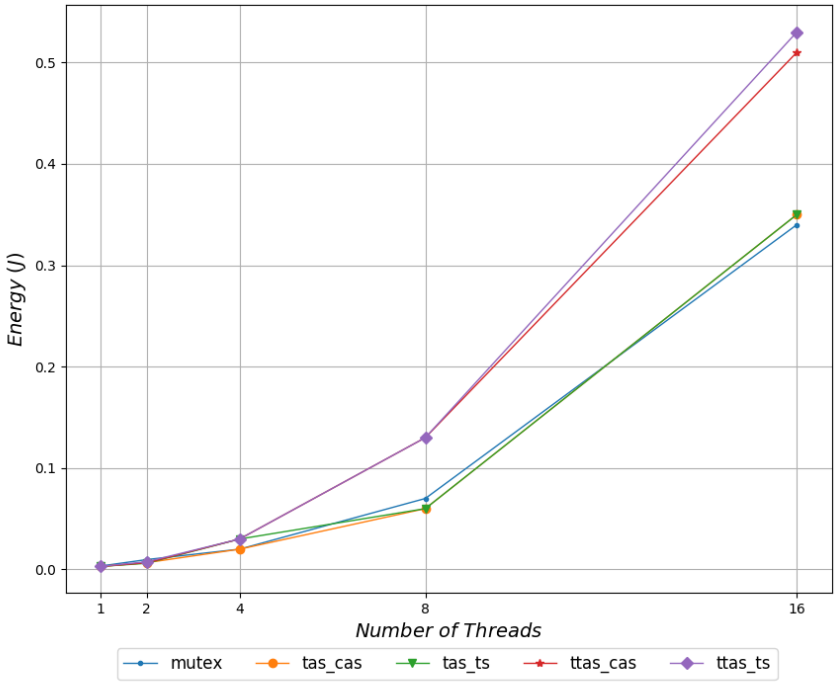
Ακολουθούν τα διαγράμματα της ενέργειας αλλά και της μετρικής EDP (Energy Delay Product) για τα προηγούμενα πειράματα:

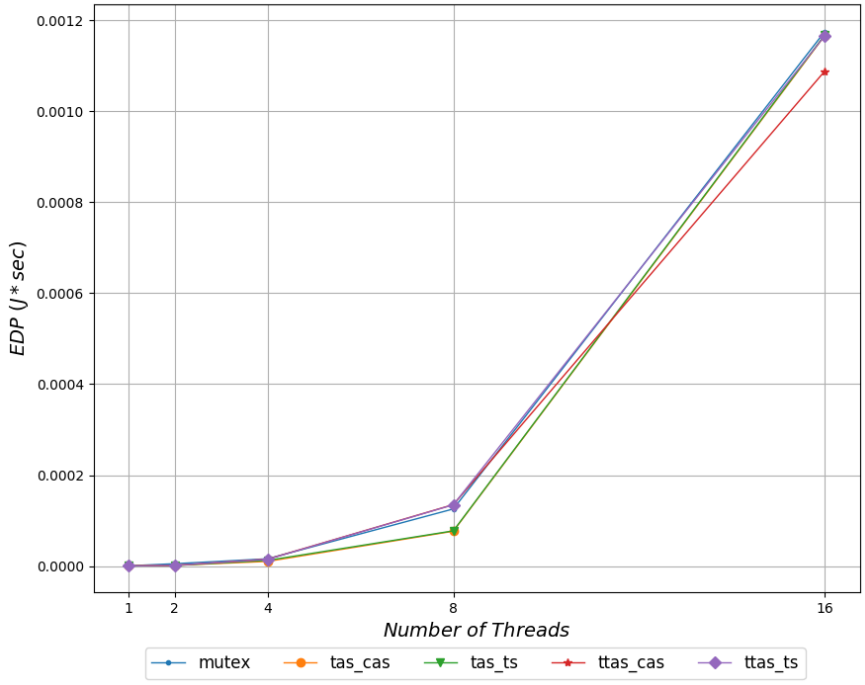
Grain Size: 1


0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Energy (J)
1
2
4
8
16
Number of Threads
mutex
tas_cas
tas_ts
ttas_cas
ttas_ts

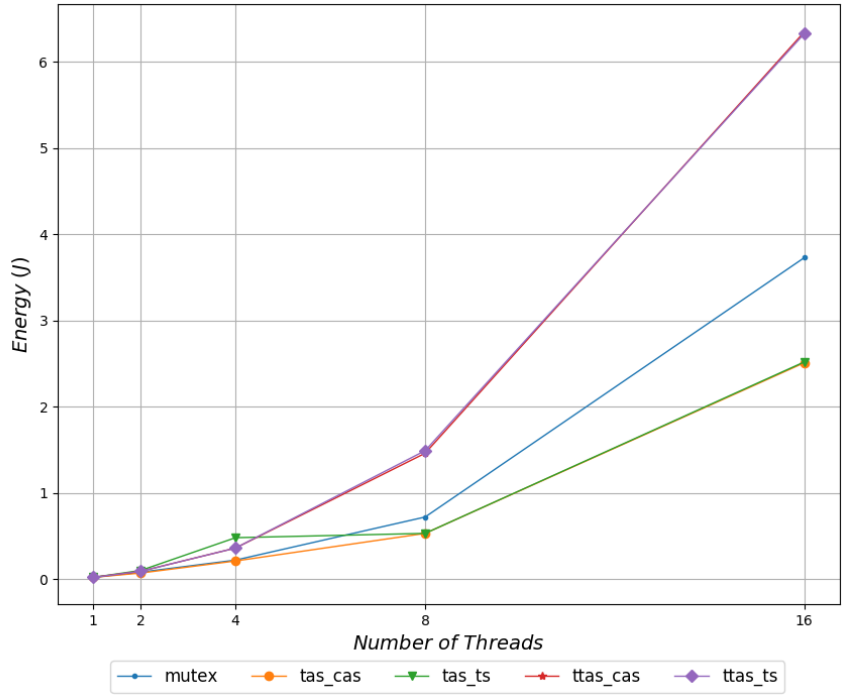

0.00020
0.00015
0.00010
0.00005
0.00000
EDP (J*sec)
1
2
4
8
16
Number of Threads
mutex
tas_cas
tas_ts
ttas_cas
ttas_ts

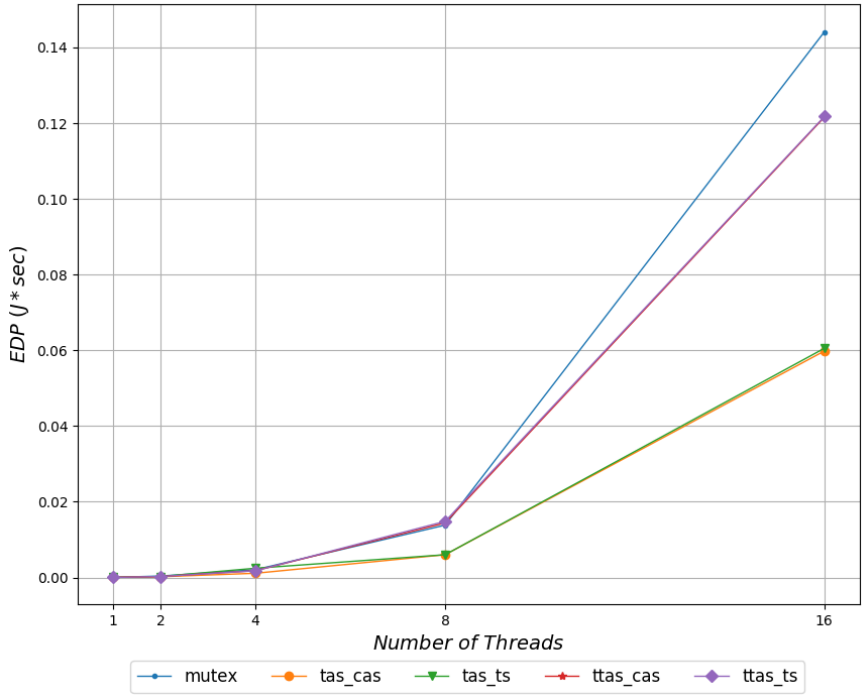
Grain Size: 10





Grain Size: 100





Σχόλια - Συμπεράσματα: Παρατηρούμε πως η ενέργεια αυξάνεται σχεδόν εκθετικά με το πλήθος των νημάτων. Ειδικότερα όταν το πλήθος αυτό γίνει από 8 νήματα σε 16 η ενέργεια που καταναλώνεται είναι σημαντικά μεγάλη.

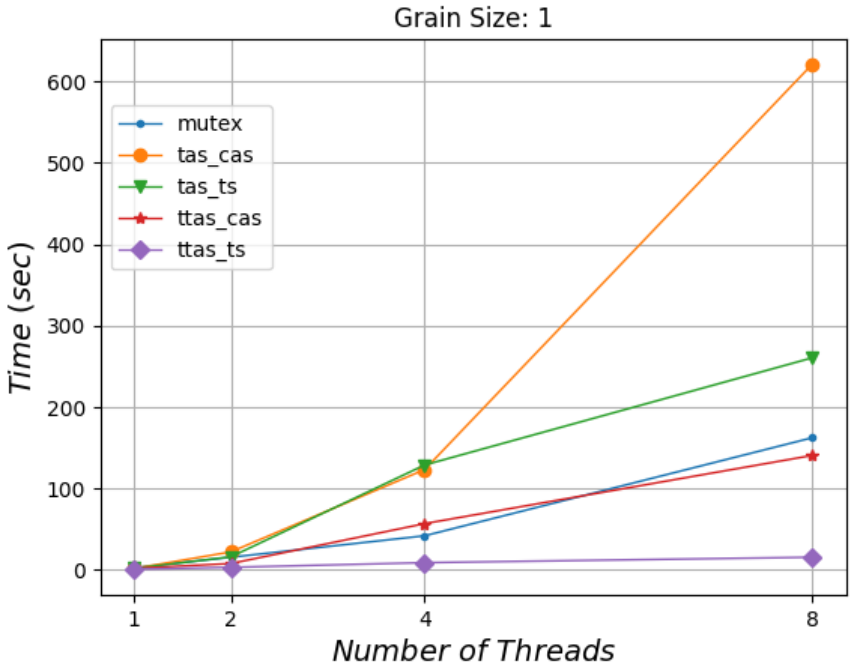
Επίσης, βλέπουμε πως για μεγαλύτερα grain size 10 και 100, οι μηχανισμοί TTAS_CAS και TTAS_TS σημειώνουν πολύ υψηλότερη κατανάλωση ενέργειας για μεγάλο αριθμό νημάτων σε σχέση με τις υπόλοιπες υλοποιήσεις. Αυτό αιτιολογείται από το γεγονός ότι κάθε νήμα που προσπαθεί να πάρει το κατειλημμένο lock κάνει τοπικά sniping πάνω στην cache του με συνεχόμενα reads τα οποία κοστίζουν ενεργειακά πολύ περισσότερο συγκριτικά με τα τα stalls εξαιτίας cache miss ή κατειλημμένο bus που εμφανίζονται στην περίπτωση των Test-and-set υλοποιήσεων. Σε όλες τις παραπάνω περιπτώσεις φθηνότερο ενεργειακά φαίνεται να είναι ο μηχανισμός TAS.

Αναφορικά με τη μετρική EDP, αξίζει να σημειώσουμε ότι για grain size 10 οι καμπύλες τείνουν να ταυτιστούν, και άρα οι μηχανισμοί μπορούν να θεωρηθούν εξίσου καλοί από πλευρά Energy Delay Product. Στη γενική περίπτωση όμως, με αυτό το κριτήριο κερδίζουν οι μηχανισμοί TAS-CAS και TAS-TS. Όσον αφορά το μηχανισμό MUTEX έχει τη χειρότερη επίδοση κρίνοντας με τη μετρική EDP για μεγάλο πλήθος νημάτων (>8), ανεξαρτήτως grain size.

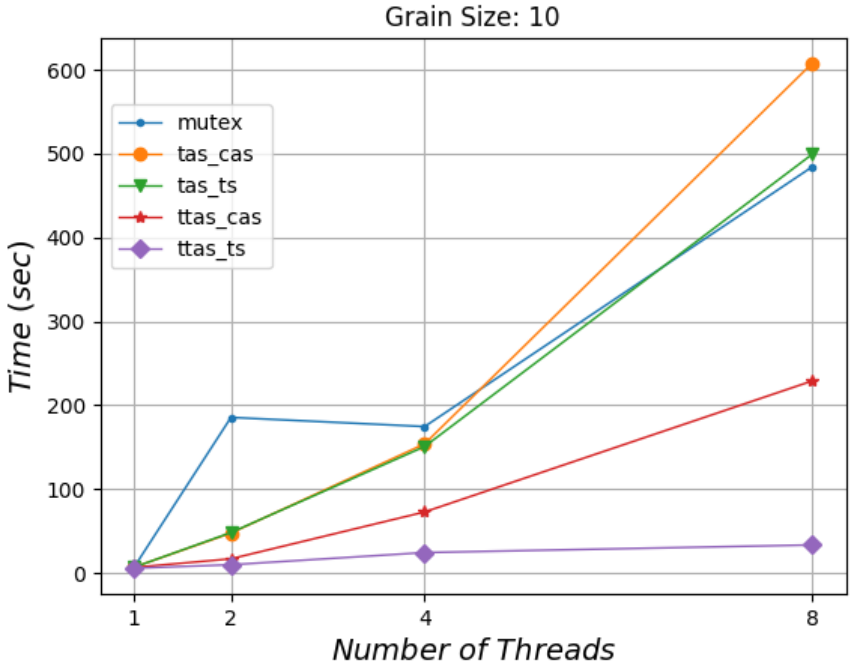
2.4 Ερώτημα 3-1-4

Εν συνεχεία τα προηγούμενα πειράματα εκτελέστηκαν σε πραγματικό σύστημα 4 physical cores με τεχνολογία hyperthreading, συνεπώς με 8 logical cores. Ο αριθμός επαναλήψεων τέθηκε ίσος με 150000000. Ακολουθούν τα διαγράμματα του χρόνου:

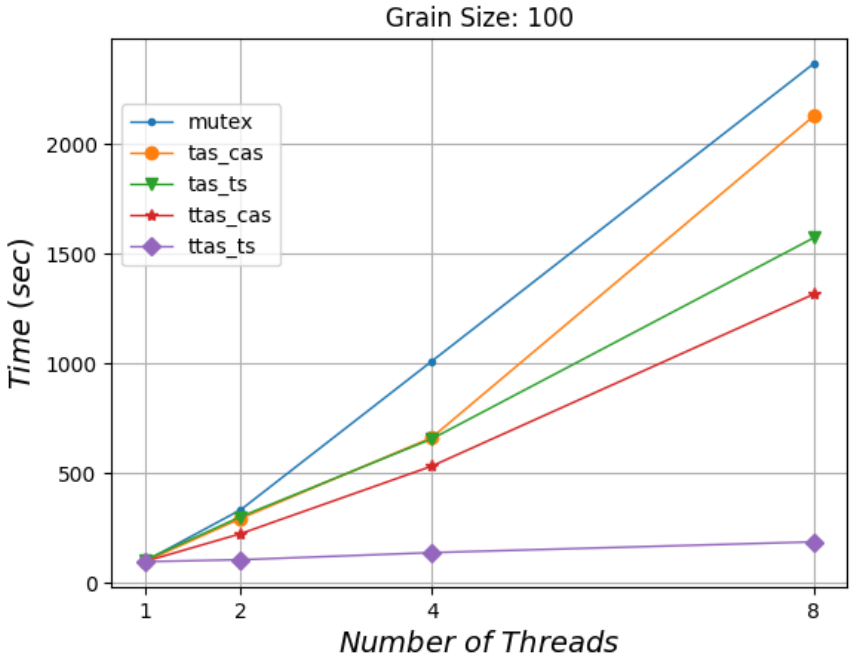
Grain Size: 1



Grain Size: 10



Grain Size: 100



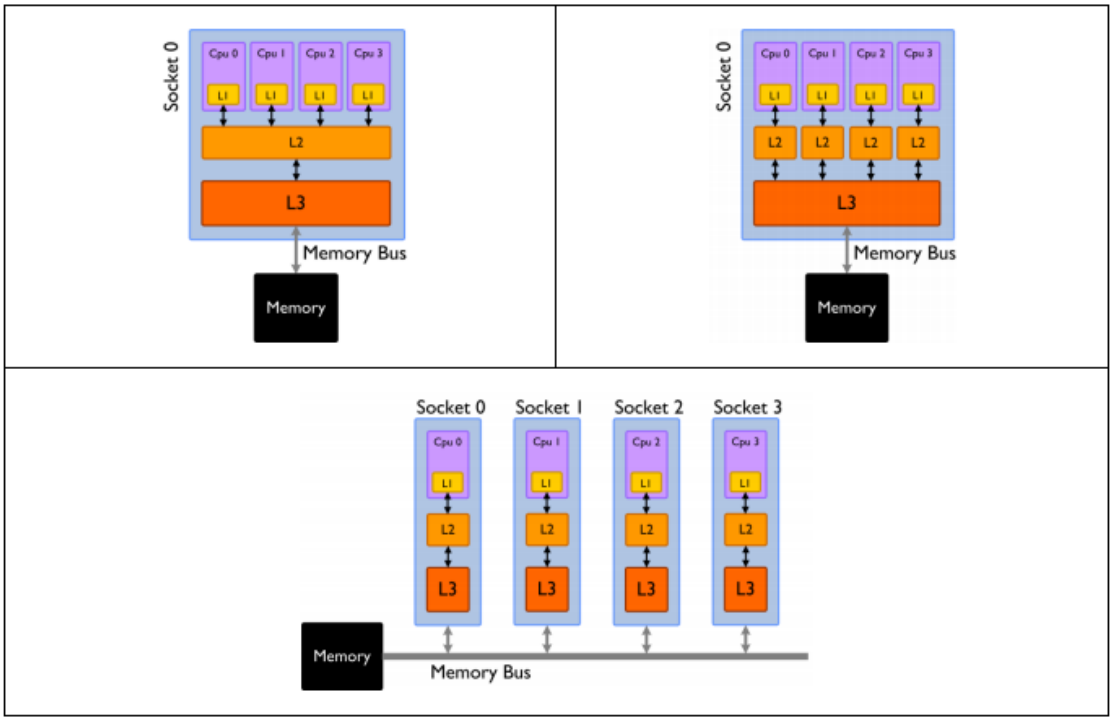
Συμπεράσματα - Σχόλια: Βλέπουμε πως κι εδώ η κλιμάκωση είναι σχεδόν γραμμική, όπως και τα αποτελέσματα των προσομοιώσεων. Αυτό που είναι αξιοσημείωτο, είναι πως την βέλτιστη επίδοση επιτυγχάνει το TTAS_TS, με πολύ μεγάλη διαφορά στην μετρική του χρόνου σε σύγκριση με τους υπόλοιπους μηχανισμούς. Μάλιστα, καθώς αυξάνεται το πλήθος των πυρήνων σχεδόν ο χρόνος εκτέλεσης παραμένει σταθερός! Την επόμενη καλύτερη επίδοση σε όλα τα grain sizes παρουσιάζει το TTAS_CAS, δηλαδή ο ίδιος μηχανισμός υλοποιημένος με Compare and Swap atomic instructions.

Είναι αξιοσημείωτο επίσης πως καθώς μεταβαίνουμε από τα 4 στα 8 νήματα, οι διαφορετικές υλοποιήσεις του μηχανισμού TAS αποκλίνουν σημαντικά στην επίδοση, με την υλοποιήση του TAS_CAS να εμφανίζει χειρότερη επίδοση από τον TAS_TS. Μάλιστα φαίνεται πως η επίδοση του TAS_CAS στις περισσότερες περιπτώσεις είναι παραπλήσια ή χειρότερη του MUTEX.

2.5 Ερώτημα 3-2

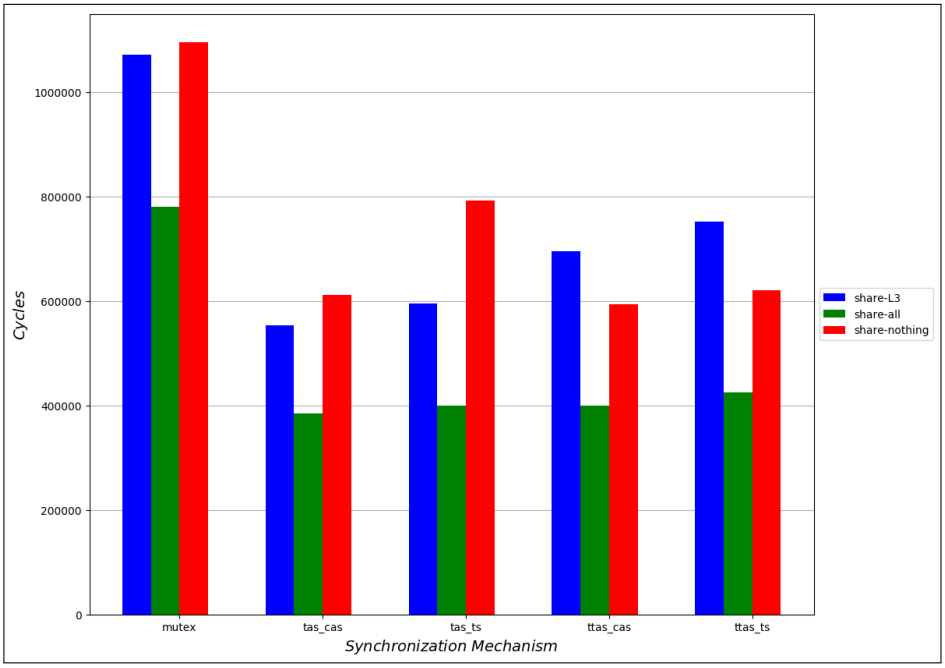
Σε αυτό το τμήμα της άσκησης εξετάζουμε την επίδοση των κάτωθι διαφορετικών τοπολογιών που διαφέρουν μεταξύ τους ως προς τον διαμοιρασμό των κρυφών μνημών. Συγκεκριμένα εξετάζουμε τις ακόλουθες τοπολογίες:

- share-all: και τα 4 νήματα βρίσκονται σε πυρήνες με κοινή L2 cache
- share-L3: και τα 4 νήματα βρίσκονται σε πυρήνες με κοινή L3 cache, αλλά όχι κοινή L2
- share-nothing: και τα 4 νήματα βρίσκονται σε πυρήνες με διαφορετική L3 cache

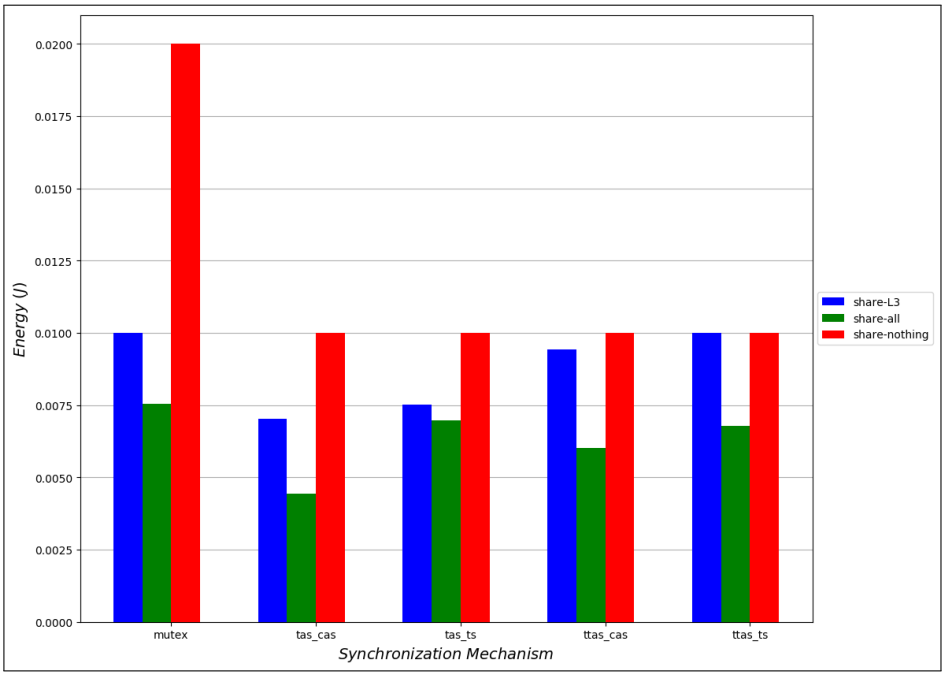


Στη συνέχεια ακολουθούν ραβδογράμματα τα οποία παρουσιάζουν τους χρόνους εκτέλεσης, την κατανάλωση ενέργειας και την μετρική EDP για τα configurations αυτά:

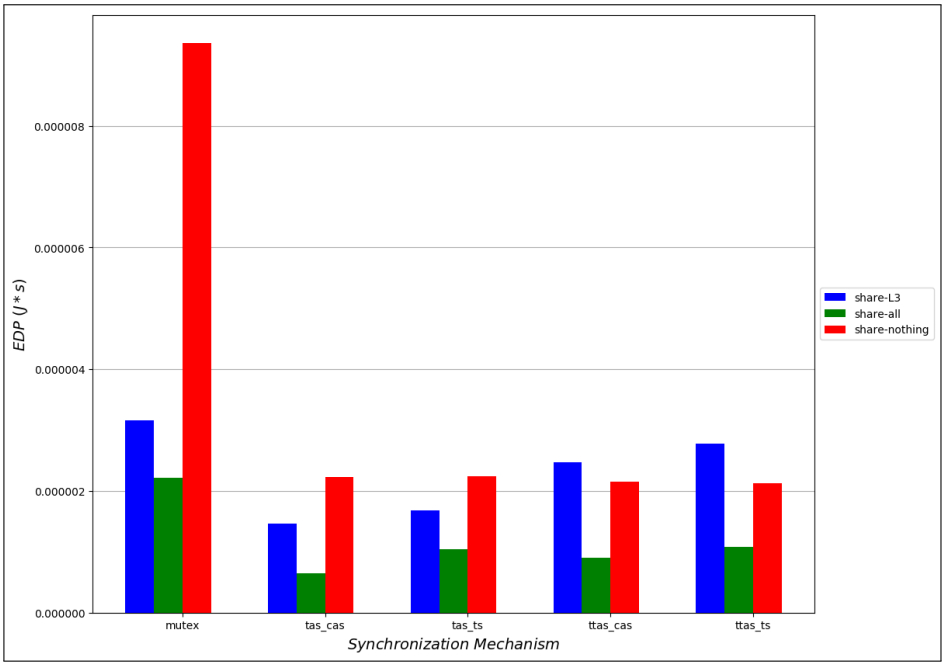
Time analysis



Energy Analysis



EDP Analysis



Συμπεράσματα - Σχόλια Όσον αφορά την επίδοση με βάση τη μετρική του χρόνου (ή αντίστοιχα κύκλος εκτέλεσης) για όλους τους υπο μελέτη μηχανισμούς συγχρονισμου, την καλύτερη επίδοση λαμβάνουμε στην τοπολογία **share-all**. Για τους μηχανισμούς συγχρονισμού MUTEX, TAS_TS, TAS_CAS οι τοπολογίες σε φθίνουσα επίδοση είναι οι share-all, share-L3, share-nothing. Για τους μηχανισμούς συγχρονισμού TTAS_TS, TTAS_CAS οι τοπολογίες σε φθίνουσα επίδοση είναι οι share-all, share-nothing, share-L3. Η βέλτιστη επίδοση του share-all αιτιολογείται από το γεγονός οτι αν επεξεργαστής ζητήσει ένα Invalid cache line θα ενημερωθεί η ιεραρχία μνήμης μέχρι την L2 και θα το διαβάσει από εκεί, ενώ στο share-L3 η invalid cache line θα ζητηθεί από την L3 και για share-nothing φτάνουμε μέχρι και την κύρια μνήμη για κάθε αυξάνοντας σημαντικά τον χρόνο που απαιτείται, αφού όσο πιο "μακριά" ιεραρχικά βρίσκεται μία μνήμη από τον επεξεργαστή τόσο πιο αργή είναι.

Ως προς την κατανάλωση ενέργειας την μεγαλύτερη κατανάλωση έχουμε στην τοπολογία share-nothing, ενώ την μικρότερη κατανάλωση στην τοπολογία share-all. Αντίστοιχα, με κριτήριο την EDP μετρική, την καλύτερη

επίδοση έχει η τπολογία share-all.

Ακολουθεί η υλοποίηση των μηχανισμών στο αρχείο lock.h:

```
1  typedef volatile int spinlock_t;
2
3  #define UNLOCKED  0
4  #define LOCKED    1
5
6  static inline void spin_lock_init(spinlock_t *spin_var) {
7      *spin_var = UNLOCKED;
8  }
9
10 static inline void spin_lock_tas_cas(spinlock_t *spin_var) {
11     while (__sync_val_compare_and_swap(spin_var, UNLOCKED, LOCKED));
12 }
13
14 static inline void spin_lock_ttas_cas(spinlock_t *spin_var) {
15     do {
16         while (*spin_var == LOCKED);
17     } while (__sync_val_compare_and_swap(spin_var, UNLOCKED, LOCKED));
18 }
19
20 static inline void spin_lock_tas_ts(spinlock_t *spin_var) {
21     while (__sync_lock_test_and_set(spin_var, LOCKED));
22 }
23
24 static inline void spin_lock_ttas_ts(spinlock_t *spin_var) {
25     do {
26         while (*spin_var == LOCKED);
27     } while (__sync_lock_test_and_set(spin_var, LOCKED));
28 }
```

3 Μέρος Β': Εκτέλεση Αλγορίθμου Tomasulo

Ακολουθεί ο πίνακας με την εκτέλεση του αλγορίθμου Tomasulo. Η στήλη No είναι ο αύξων αριθμός κάθε γραμμής του πίνακα, ενώ η στήλη CMD δίνει τον αριθμό (γραμμή - ξεκινώντας από το 0) της εντολής στο αρχικό κομμάτι κώδικα που μας δίνεται. Οι σειρές που χρωματίζονται με κίτρινο χρώμα αντιστοιχούν σε εντολές οι οποίες γίνονται flush.

No	CMD	OP	IS	EX	WR	CMT	Σχόλιο
0	0	LD F0, 0(R1)	1	2-5	6	7	Miss on A[0] - fetch(A[0], A[1])
1	1	ADDD F4, F4, F0	1	7-9	10	11	RAW(F0)
2	2	LD F1, 0(R2)	2	3-6	7	11	Miss on B[0] - fetch(B[0], B[1]) - New Cache content: A[0],A[1].B[0],B[1]
3	3	MULD F4, F4, F1	2	11-15	16	17	RAW(F1, F4)
4	4	ANDI R9, R8, 0x2	3	4-5	8	17	R9=0 - CDB Conflict για 6ο και 7ο κύκλο, άρα WR στον 8ο κύκλο
5	5	BNEZ R9, NEXT	3	9-10	11	18	RAW(R9) - BHR=0 - ADDR: 0x0044843C, FSM=11, predicts T, res=NT
6	9	LD F5, 8(R1)	7	8	9	-	Hit A[1], Load Queue Full till cycle 6
7	10	ADDD F4, F4, F5	7	-	-	-	RAW(F4), το F4 έτοιμο στον κύκλο 16, όμως ήδη στον 11ο θα έχει γίνει fush
8	11	ADDI R1, R1, 0x8	8	9-10	-	-	flush @cycle 11
9	12	SUBI R8, R8, 0x1	9	10-11	-	-1	Reservation Stations full till cycle 8 - Η επόμενη εντολή BNEZ δε θα βρει RS πριν τον κυκλο 12 αρα δεν γίνεται ποτέ issue - flush @cycle 11
10	6	LD F2, 16(R2)	12	13-16	17	18	Cache Miss, fetch(B[2],B[3]), New Cache Content A[0], A[1], B[2], B[3]
11	7	MULD F2, F2, F5	12	18-22	23	24	RAW(F2)
12	8	ADDD F4, F4, F2	13	24-26	27	28	RAW(F2)
13	9	LD F2, 16(R2)	13	14	15	28	Cache Hit B[2]
14	10	ADDD F4, F4, F5	14	28-30	31	32	RAW(F4)
15	11	ADDI R1, R1, 0x8	14	15-16	18	32	CDB
16	12	SUBI R8, R8, 0x1	18	19-20	21	33	RoB full till 17 cycle
17	13	BNEZ R8, LOOP	18	22-23	24	33	FSM=0; predicts T, res=NT, RAW(R8), θα γίνει flush στον 24ο κύκλο
18	0	LD F0, 0(R1)	19	20	22	-	Cache Hit A[1], CDB
19	1	ADDD F4, F4, F0	19	-	-	-	RAW(F4), θα είναι γνωστό στον 31κύκλο. Με την εντολή αυτή γεμίζει ο RoB - Διαθέσιμη θέση θα υπάρξει ξανά μόλις απελευθερωθεί θέση στο POB, στον 25ο κυκλο! Τότε θα έχει γίνει ήδη flush
20	14	SD F4, 8(R2)	25	32-35	36	37	RAW(F4), Cache Miss