Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Αρχιτεκτονική Υπολογιστών

Branch Prediction

Χαράλαμπος Παπαδόπουλος
03120199

Προηγμένα Θέματα Αρχιτεκτονικής Υπολογιστών
1η Άσκηση

Απρίλιος 2025

1 Εισαγωγή

Στα πλαίσια της παρούσας άσκησης θα χρησιμοποιήσουμε το εργαλείο PIN για να μελετήσουμε την επίδραση διαφορετικών συστημάτων πρόβλεψης εντολών άλματος, καθώς και την αξιολόγησή τους με δεδομένο το διαθέσιμο χώρο πάνω στο τσιπ.

Στη ροή εκτέλεσης ενός προγράμματος, οι εντολές διακλάδωσης (branches) παίζουν κρίσιμο ρόλο, καθώς καθορίζουν την πορεία που θα ακολουθήσει η εκτέλεση βάσει συνθηκών και δομών ελέγχου (όπως if, for, while, κ.ά.). Οι διακλαδώσεις μπορεί να είναι υποθετικές (conditional), όπου η λήψη της απόφασης εξαρτάται από μια συνθήκη, ή ανεπιφύλακτες (unconditional), όπου η μετάβαση είναι δεδομένη.

Η πρόβλεψη διακλαδώσεων (branch prediction) είναι μία τεχνική που εφαρμόζεται σε σύγχρονους επεξεργαστές για να μειώσει την καθυστέρηση που προκαλείται από την αβεβαιότητα της εκτέλεσης τέτοιων εντολών. Οι σύγχρονοι επεξεργαστές βασίζονται σε σωληνώσεις (pipelines) και εκτέλεση πολλαπλών εντολών παράλληλα (superscalar execution), καθιστώντας τη σωστή πρόβλεψη των διακλαδώσεων κρίσιμη για τη διατήρηση της απόδοσης.

Η αποδοτικότητα ενός μηχανισμού πρόβλεψης επηρεάζει άμεσα την ταχύτητα και την ενεργειακή κατανάλωση ενός συστήματος. Ένας λανθασμένος προγνωστικός μηχανισμός μπορεί να οδηγήσει σε καθυστερήσεις, καθώς απαιτείται εκκαθάριση και ανανέωση του pipeline. Για τον λόγο αυτό, η επιλογή και αξιολόγηση της κατάλληλης στρατηγικής πρόβλεψης, ανάλογα με τον διαθέσιμο χώρο και την πολυπλοκότητα του υλικού, αποτελεί σημαντικό αντικείμενο μελέτης στην αρχιτεκτονική υπολογιστών.

2 Ανάλυση εντολών άλματος

Αρχικά, συλλέγουμε δεδομένα σχετικά με τα benchmarks που θα χρησιμοποιήσουμε (τόσο για τα train όσο και τα ref) χρησιμοποιώντας το pintool cslab_branch_stats.so. Λαμβάνουμε, λοιπόν τα εξής αποτελέσματα:

Benchmark	Total Branches	Conditional Taken	Conditional Not Taken	Unconditional Branches	Calls	Returns
401.bzip2	48731240680	14828890244	29233486500	4503343406	82760267	82760263
403.gcc	754261098	269641225	304818852	66609945	56595540	56595536
410.bwaves	31873093532	21133244425	8406203291	1791546316	271049752	271049748
416.gamess	14348669381	6258963398	6983826386	863109166	121385219	121385212
429.mcf	3862679407	1332802908	2377832566	97772747	27135595	27135591
433.milc	2240722314	1242012269	541178208	21600368	217965738	217965731
435.gromacs	18177458459	5651211656	11410526337	778103616	168808427	168808423
436.cactusADM	170600508	142370779	19042164	1719499	3734035	3734031
437.leslie3d	17218327107	16400961590	810904801	4541911	959406	959399
450.soplex	1261449374	640242394	484809667	46077417	45159950	45159946
456.hmmer	13914144105	8669664400	4743905013	94489468	203042614	203042610
459.GemsFDTD	3408454302	2895485194	348903461	50019703	57022974	57022970
464.h264ref	48076004067	24275875054	8491727313	3326070462	5991165621	5991165617
470.lbm	1356406164	176492622	922282496	252368694	2631178	2631174
471.omnetpp	50858988954	11220928577	21007505135	4549937533	7040308988	7040308721
483.xalancbmk	54702087725	8947195710	19089680931	5016363416	10824423836	10824423832

Table 1: Branch Prediction Statistics for Train Benchmarks

Benchmark	Total Branches	Conditional Taken	Conditional Not Taken	Unconditional Branches	Calls	Returns
401.bzip2	45686521905	15926493292	22623211331	4456192564	1340312361	1340312357
403.gcc	33841317469	12296277104	13569715817	3134043792	2420640380	2420640376
410.bwaves	128568449445	88592484629	34014693916	3853748332	1053761286	1053761282
416.gamess	73894102026	34756617371	32132851363	5481923933	761354687	761354672
429.mcf	67874046952	25764447757	39087004417	2224162736	399216023	399216019
433.milc	67919982540	36675874082	16584640014	1322820263	6668324094	6668324087
435.gromacs	84256714798	26179759996	52980568742	3595806252	750289906	750289902
436.cactusADM	4368886354	4156985961	165208684	8975841	18857936	18857932
437.leslie3d	119714175105	114398465011	5298773071	14378054	1279488	1279481
450.soplex	57636398865	31511622629	20460639260	1923982890	1870077045	1870077041
456.hmmer	42165181169	27398563731	14228668166	221630014	158159631	158159627
459.GemsFDTD	47114377011	33969941316	7764020378	2457796415	1461309453	1461309449
464.h264ref	48076004019	24275874976	8491727354	3326070449	5991165622	5991165618
470.lbm	15278518783	527480913	11071877357	3673870653	2644932	2644928
471.omnetpp	137784824709	29163496937	63903123884	11932487679	16392858237	16392857972
483.xalancbmk	287764730034	75393414336	162510582834	8390230048	20735251410	20735251406

Table 2: Branch Prediction Statistics for Ref Benchmarks

3 N-bit predictors

Σε αυτό το ερώτημα εξετάζουμε την ανάλυση κάποιων branch predictors. Οι N-bit predictors χρησιμοποιούν περισσότερα από ένα bit για να παρακολουθήσουν την ιστορία των αποφάσεων των branch. Αντί να χρησιμοποιούν μόνο ένα bit για να καταγράψουν αν ένα branch εκτελείται ή όχι, χρησιμοποιούν έναν μετρητή που αυξάνεται ή μειώνεται με βάση τις προηγούμενες αποφάσεις. Αυτή η μέθοδος βοηθάει στην καλύτερη πρόβλεψη, καθώς καταγράφει πιο πολύπλοκες συμπεριφορές των branch.

Για παράδειγμα, ένας 2-bit predictor χρησιμοποιεί έναν μετρητή που μπορεί να έχει 4 καταστάσεις, με τις οποίες μπορεί να προβλέψει με μεγαλύτερη ακρίβεια τη συμπεριφορά ενός branch, καθώς χρειάζονται δύο συνεχόμενα λάθη για να αλλάξει η πρόβλεψη. Η χρήση περισσότερων bit (όπως 3-bit ή 4-bit) βελτιώνει ακόμη περισσότερο την ακρίβεια, αλλά απαιτεί περισσότερη μνήμη και υπολογιστική ισχύ.

3.1

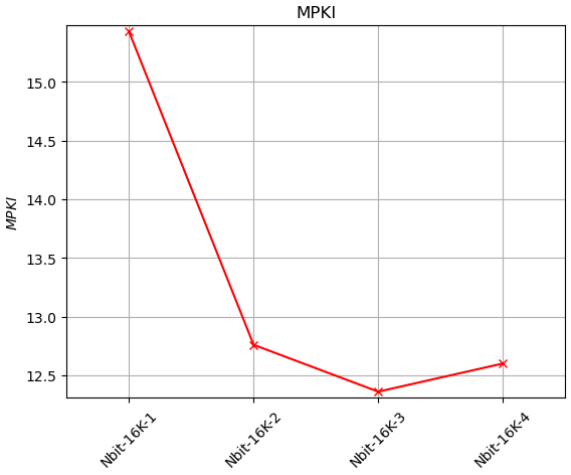
Διατηρώντας σταθερό τον αριθμό των BHT entries και ίσο με 16K, προσομοιώνουμε τους n-bit predictors, για N = 1, 2, 3, 4. Τα n-bits υλοποιούν ένα saturating up-down counter (cslab_branch.cpp) όπως είδαμε στις διαλέξεις. Τροποιούμε κατάλληλα τον βοηθητικό κώδικα προσθέτοντας τον εξής κώδικα

```
for (int i=1; i <= 4; i++) {
    NbitPredictor *nbitPred = new NbitPredictor(14, i);
    branch_predictors.push_back(nbitPred);
}
```

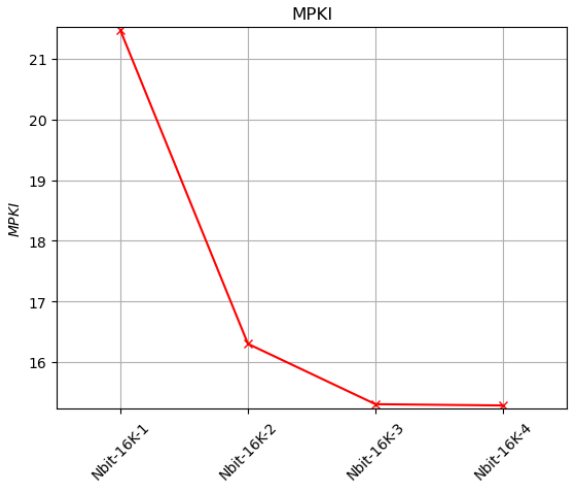
Οι παράμετροι που δίνουμε στο object `NbitPredictor` είναι:

- `index_bits = 14` : Ορίζει το πλήθος των bits που χρησιμοποιούνται για την κατασκευή του πίνακα προβλέψεων. Δηλαδή, το μέγεθος του πίνακα θα είναι 2^{14} καταχωρήσεις.
- `cntr_bits` : Ορίζει το πλήθος των bits κάθε μετρητή στον πίνακα. Όσο περισσότερα bits, τόσο περισσότερες καταστάσεις μπορεί να περιγράψει ένας μετρητής.

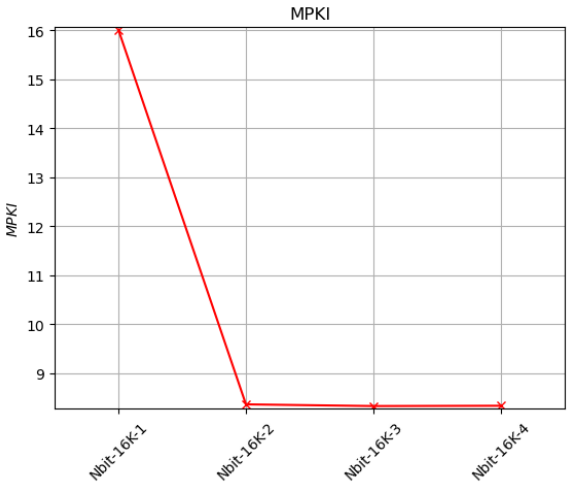
Καταλήγουμε, λοιπόν, με τα εξής διαγραμμάτα:



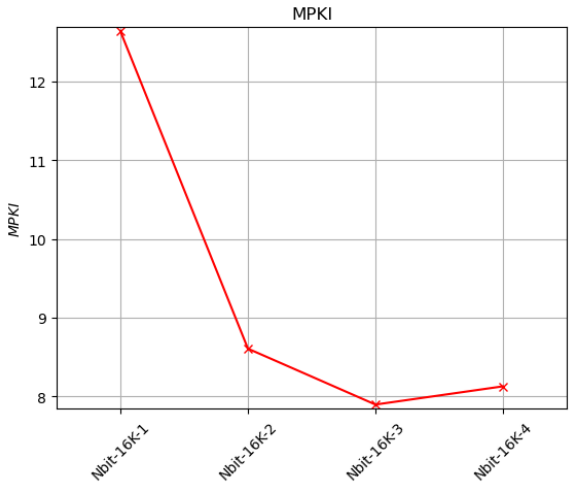
(a) 401.bzip2



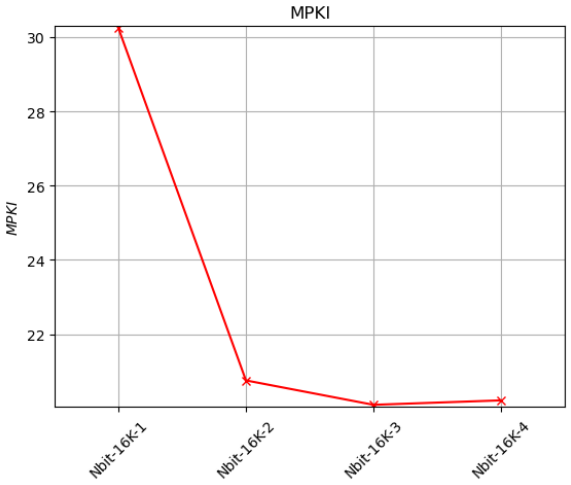
(b) 403.gcc



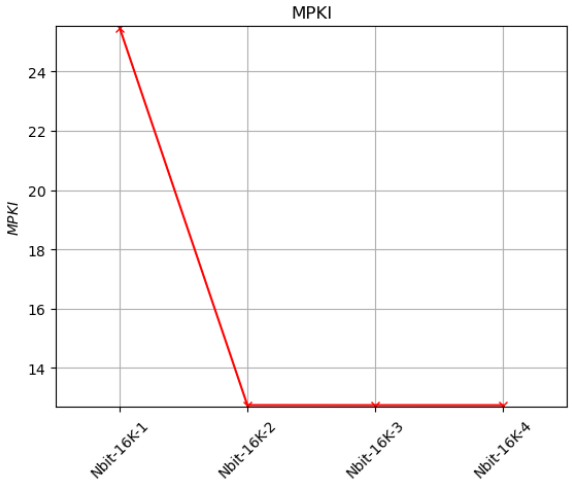
(c) 410.bwaves



(d) 416.gamess

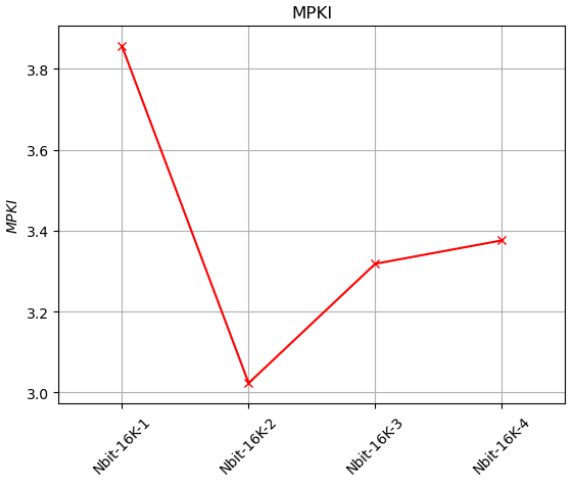


(e) 429.mcf

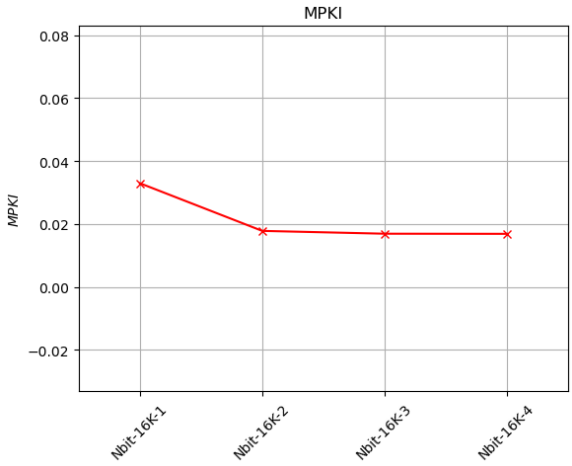


(f) 433.milc

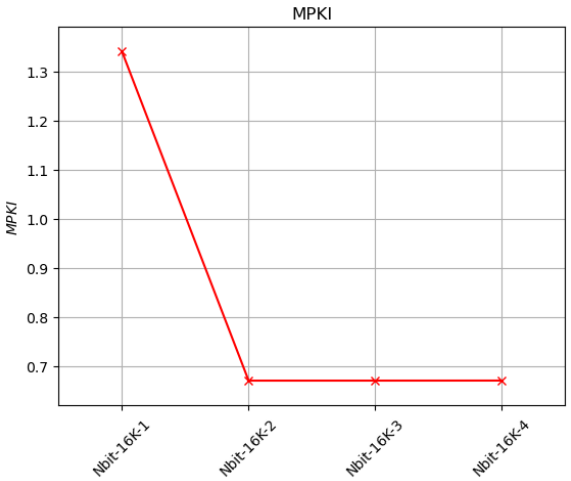
Figure 1: Branch Prediction Accuracy for Different N-bit Predictors



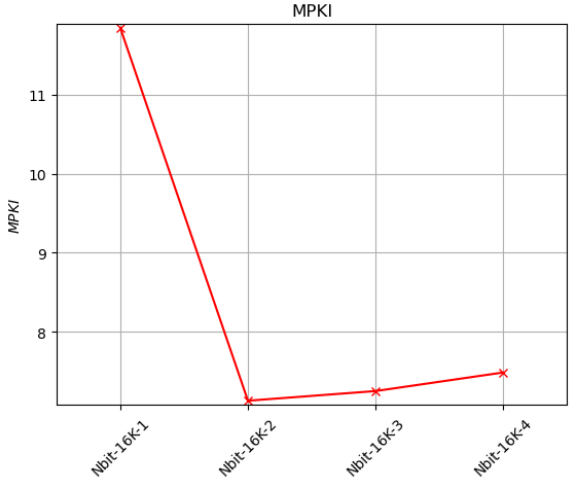
(g) 435.gromacs



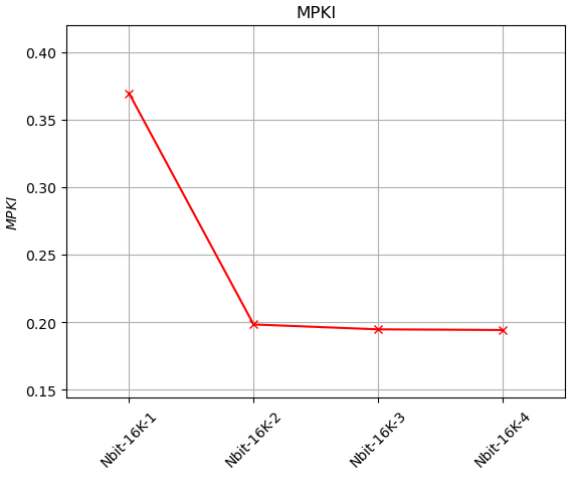
(h) 436.cactusADM



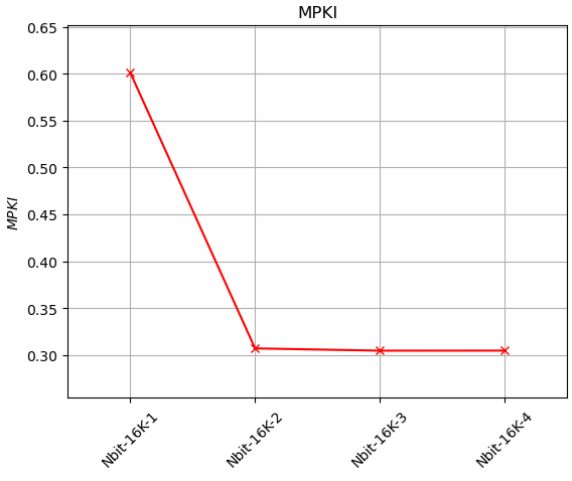
(i) 437.leslie3d



(j) 450.soplex

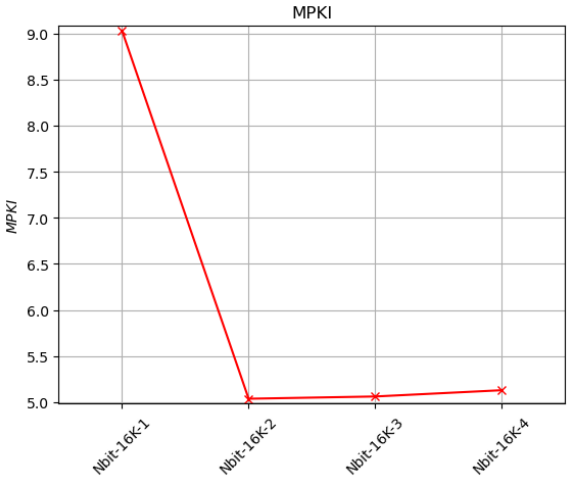


(k) 456.hmmer

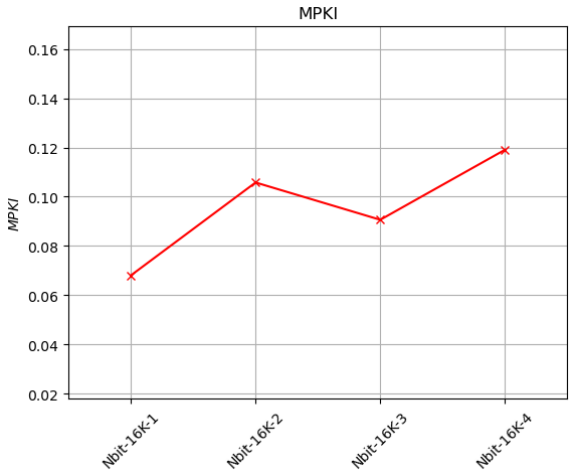


(l) 459.GemsFDTD

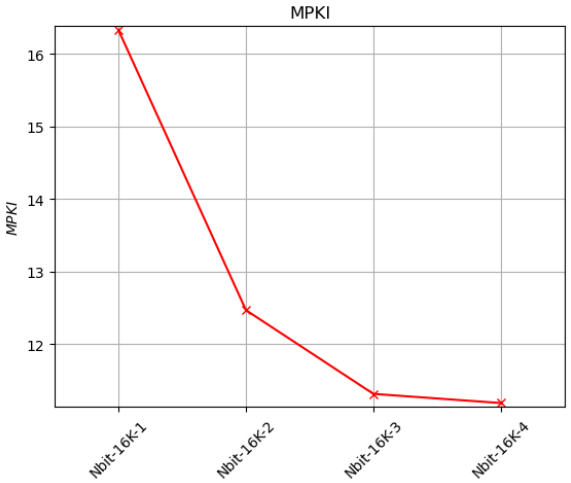
Figure 1: (continued)



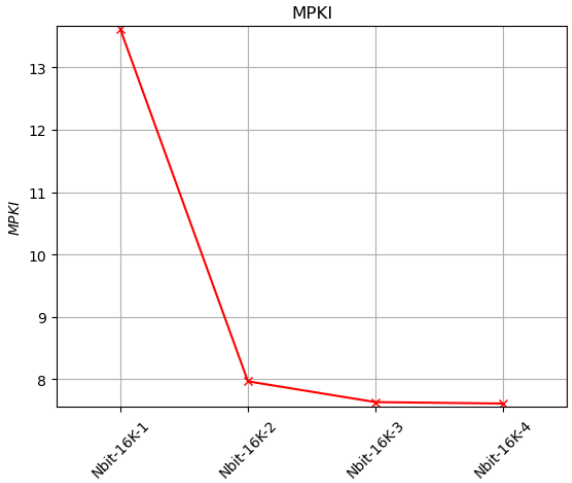
(m) 464.h264ref



(n) 470.lbm



(o) 471.omnetpp



(p) 483.xalancbmk

Figure 1: (continued)

Υπολογίζουμε τον γεωμετρικό μέσο όρο του MPKI για κάθε benchmark και για κάθε n-bit predictor.

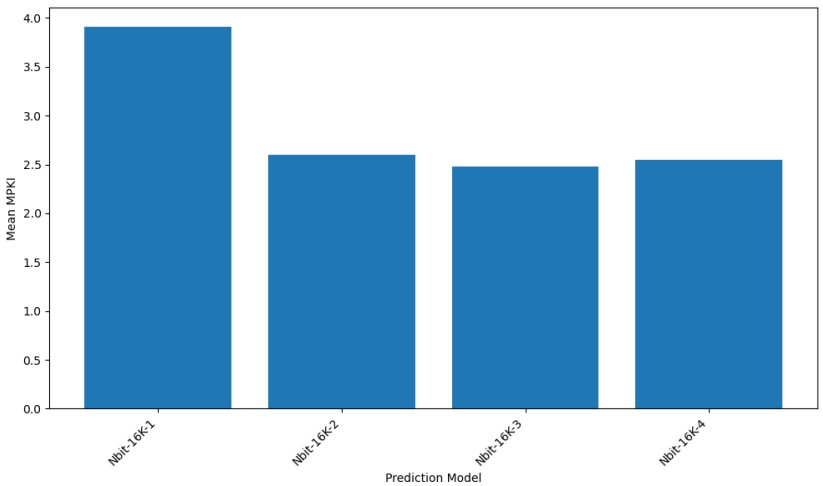


Figure 2: Geometric Mean MPKI for N-bit predictors

Παρατηρούμε από τα αποτελέσματα ότι ο Nbit-16K-1 predictor παρουσιάζει με διαφορά το υψηλότερο MPKI (3.90), ενώ οι υπόλοιποι τρεις (Nbit-16K-2, 3 και 4) κινούνται σε παρόμοια επίπεδα απόδοσης (2.60, 2.48 και 2.55 αντίστοιχα).

Επιλέγουμε τον **Nbit-16K-2**, καθώς προσφέρει ικανοποιητικές επιδόσεις με **χαμηλές απαιτήσεις σε επιπλέον υλικό (overhead)**. Αν και οι Nbit-16K-3 και Nbit-16K-4 εμφανίζουν ελαφρώς καλύτερο MPKI, η διαφορά είναι μικρή και δεν δικαιολογεί την αυξημένη πολυπλοκότητα.

3.2

Από το paper *"Optimal 2-Bit Branch Predictors"* (R. Nair, 1995) βλέπουμε τα εξής πιθανά FSM:

TABLE VI
BEST FIVE MACHINES IGNORING STARTING STATE

Rank	State diagram for machine
1.	
2.	
3.	
4.	
5.	

Figure 3: FSMs according to R. Nair.

όπου με **bold** απεικονίζονται οι μεταβάσεις μετά από σωστή πρόβλεψη και dotted οι μεταβάσεις ύστερα από αποτυχημένη.

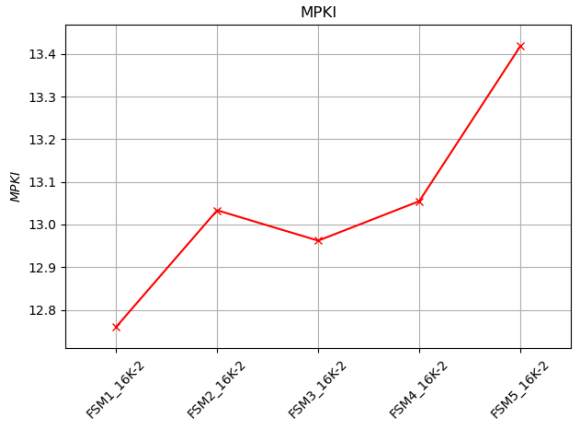
Βασιζόμενοι σε αυτά τα διαγράμματα δημιουργούμε τους αντίστοιχους predictors στο αρχείο branch_predictor.h:

```
class TwobitPredictor_FSM2 : public BranchPredictor
{
public:
    TwobitPredictor_FSM2(unsigned index_bits_ = 14, unsigned cntr_bits_ = 2)
    \\same as before ...

    virtual void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target) {
        unsigned int ip_table_index = ip % table_entries;
        if (actual) {
            if (TABLE[ip_table_index] < COUNTER_MAX)
                TABLE[ip_table_index]++;
        } else {
            if (TABLE[ip_table_index] == 2)
                TABLE[ip_table_index] -= 2;
            else if (TABLE[ip_table_index] > 0)
                TABLE[ip_table_index]--;
        }

        updateCounters(predicted, actual);
    };
```

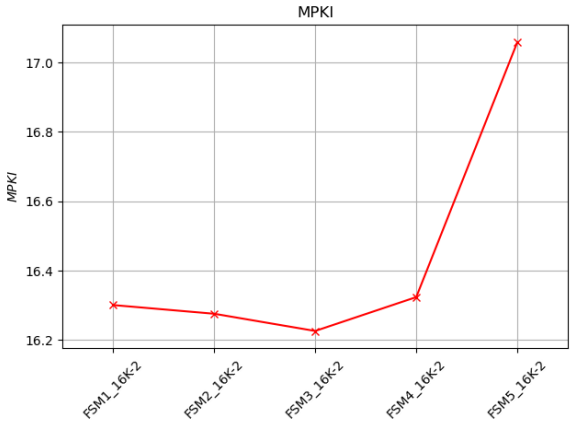
Κατά αντίστοιχο τρόπο κατασκευάζουμε και τις υπόλοιπες κλάσεις τις οποίες μετά καλούμε στο cslab_branch.cpp

```
new TwobitPredictor_FSM1();
new TwobitPredictor_FSM2();
new TwobitPredictor_FSM3();
new TwobitPredictor_FSM4();
new TwobitPredictor_FSM5();
```

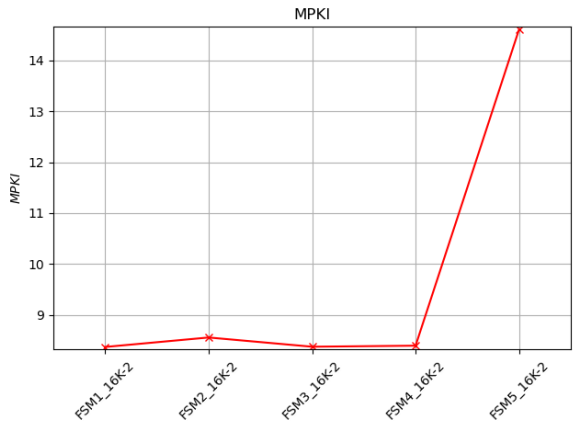
Καταλήγουμε, λοιπόν, με τα εξής διαγραμμάτα:



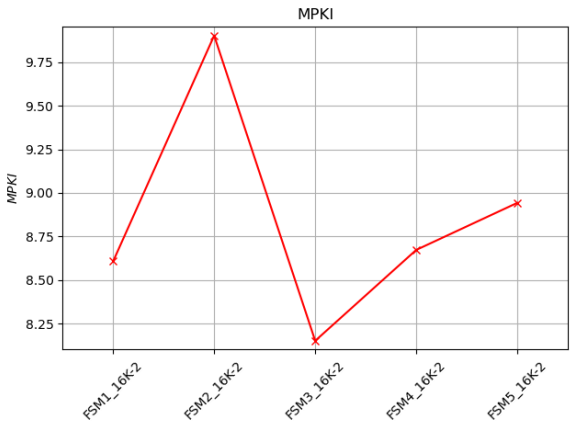
(a) 401.bzip2



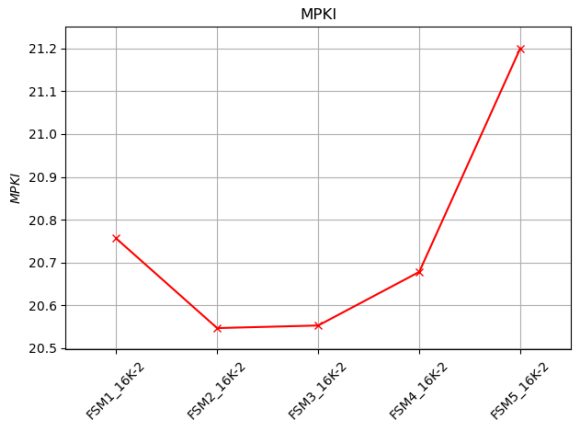
(b) 403.gcc



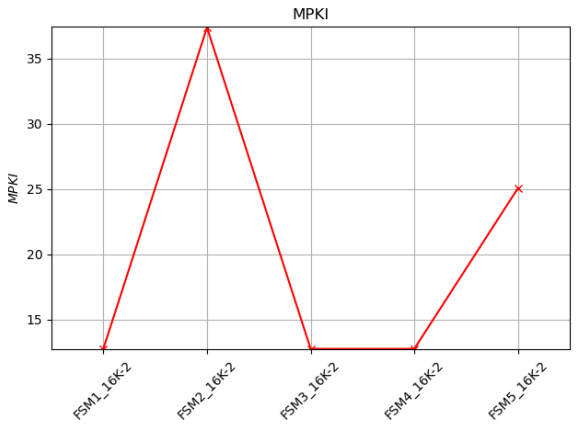
(c) 410.bwaves



(d) 416.gamess

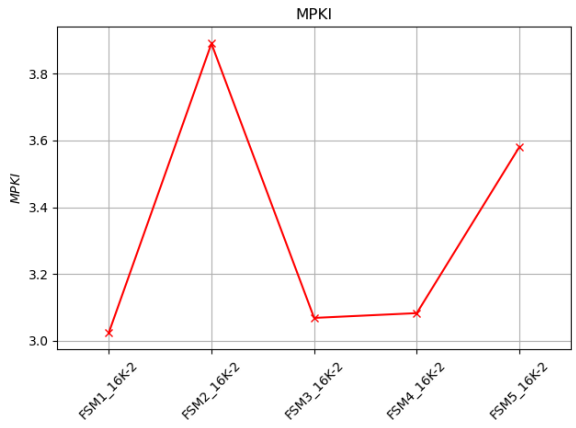


(e) 429.mcf

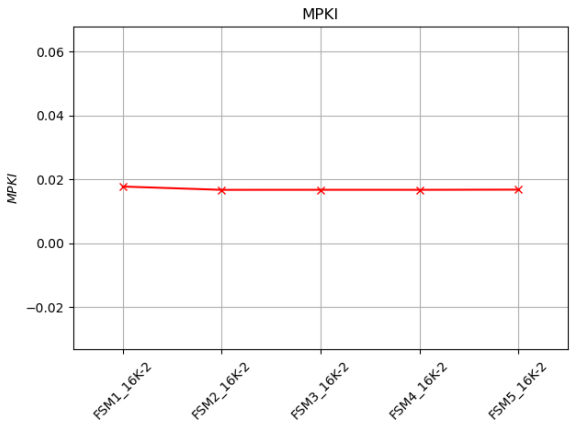


(f) 433.milc

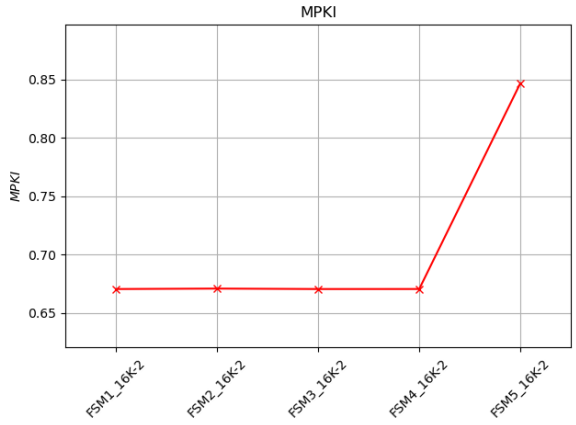
Figure 4: Branch Prediction Accuracy for Different N-bit Predictors



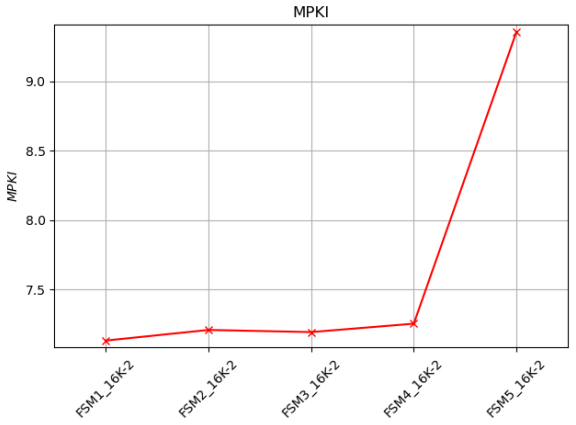
(g) 435.gromacs



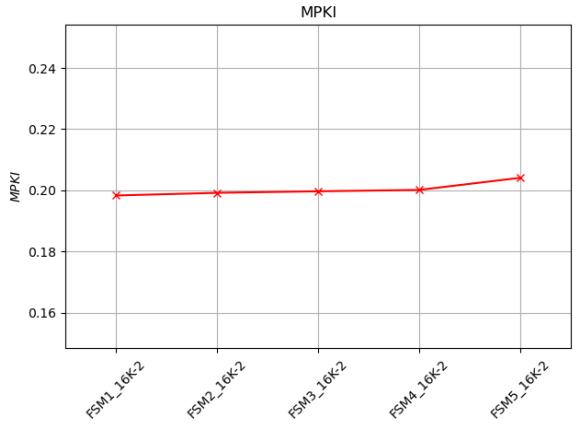
(h) 436.cactusADM



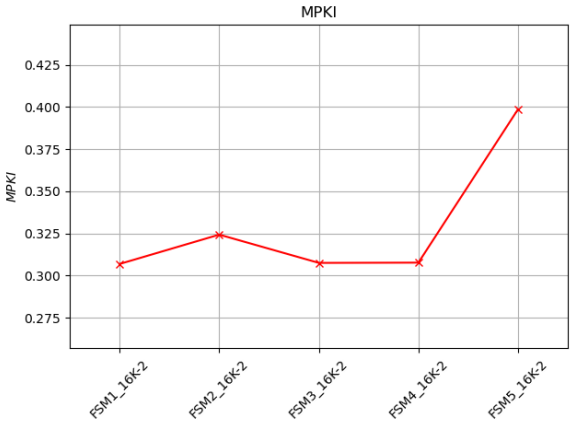
(i) 437.leslie3d



(j) 450.soplex

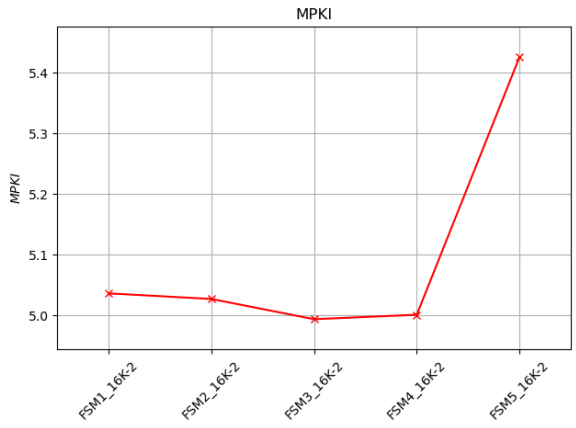


(k) 456.hmmer

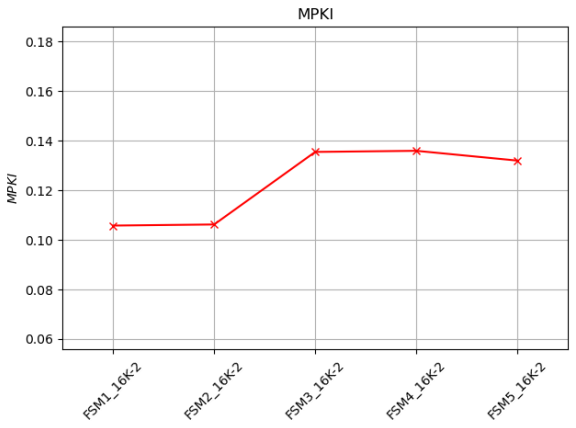


(l) 459.GemsFDTD

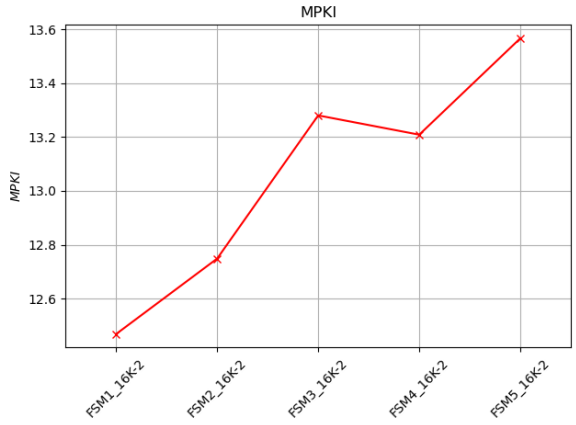
Figure 4: (continued)



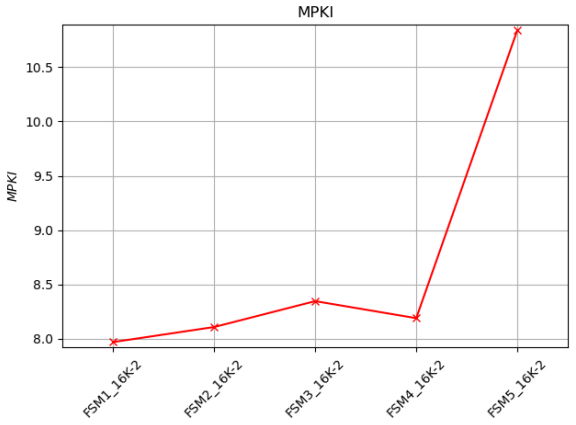
(m) 464.h264ref



(n) 470.lbm



(o) 471.omnetpp



(p) 483.xalancbmk

Figure 4: (continued)

Υπολογίζουμε τον γεωμετρικό μέσο όρο του MPKI για κάθε benchmark και για κάθε FSM model.

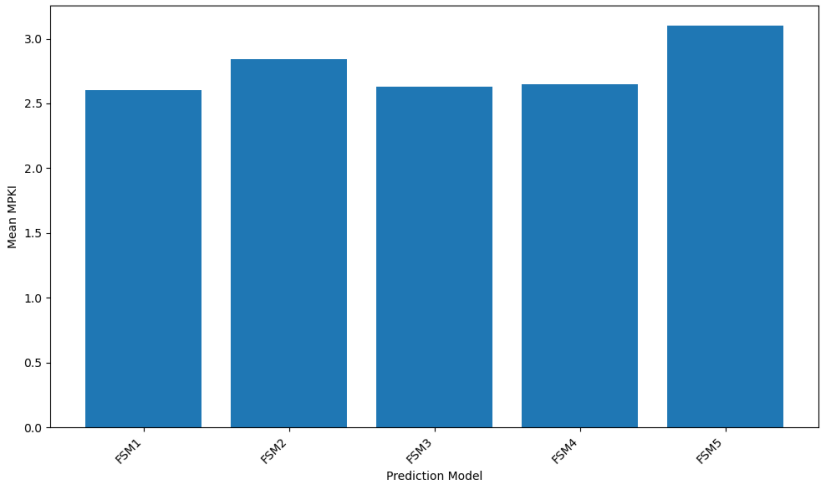


Figure 5: Geometric Mean MPKI for 16K-2 FSM models.

Επιλέγουμε και πάλι τον απλό **Nbit-16K-2** predictor με **FSM1**, καθώς προσφέρει τις **βέλτιστες επιδόσεις** μεταξύ των εξεταζόμενων FSM και υλοποιείται με σχετικά απλό τρόπο, χρησιμοποιώντας έναν απλό saturating up-down counter.

3.3

Προκειμένου να ορίσουμε το διαθέσιμο **hardware σε 32K bits**, θα πρέπει να ικανοποιείται η σχέση:

$$\texttt{index\_bits} \times \texttt{cntr\_bits} = 32K$$

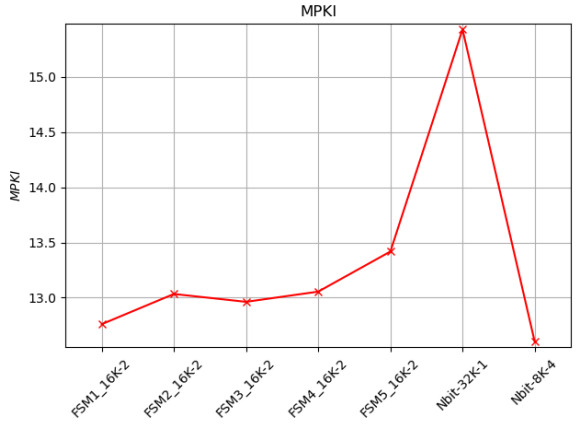
Συνεπώς, τα δυνατά ζεύγη τιμών είναι:

$$(15, 1), \quad (14, 2), \quad (13, 3)$$

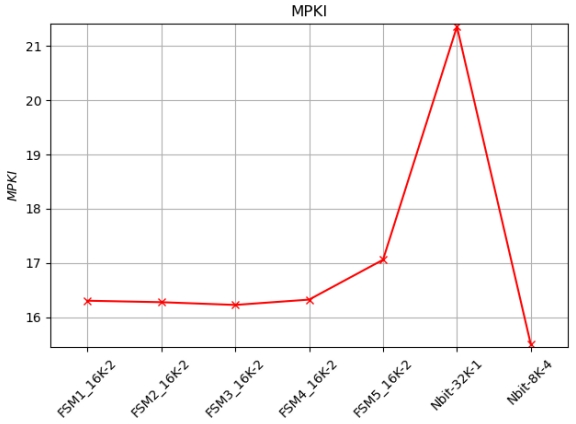
όπου το πρώτο στοιχείο κάθε ζεύγους αντιστοιχεί στον αριθμό των index bits και το δεύτερο στον αριθμό των bits του καταχωρητή (counter). Καλούμε, λοιπόν, τους εξής constructors στο cslab_branch.cpp:

```
// 32K hardware
new TwobitPredictor_FSM1();
new TwobitPredictor_FSM2();
new TwobitPredictor_FSM3();
new TwobitPredictor_FSM4();
new TwobitPredictor_FSM5();

new NbitPredictor(15, 1);
new NbitPredictor(13, 4);
```

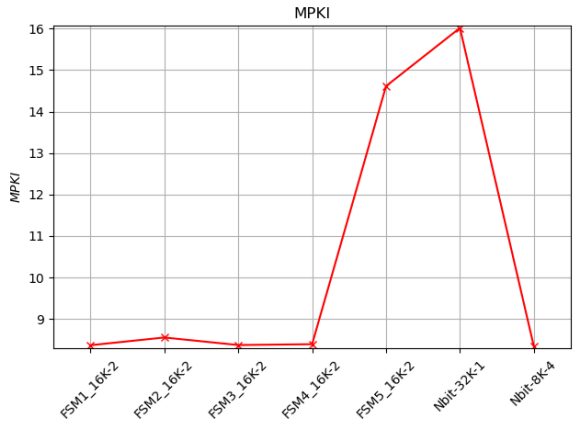
Και καταλήγουμε με τα εξής διαγράμματα:



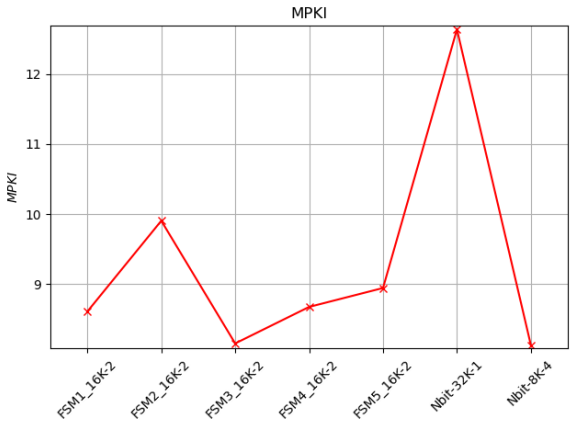
(a) 401.bzip2



(b) 403.gcc

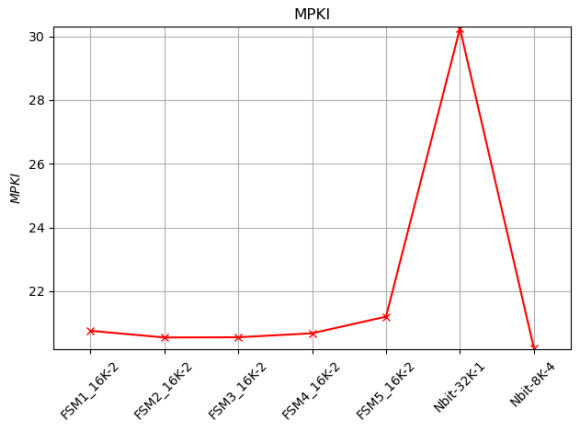


(c) 410.bwaves

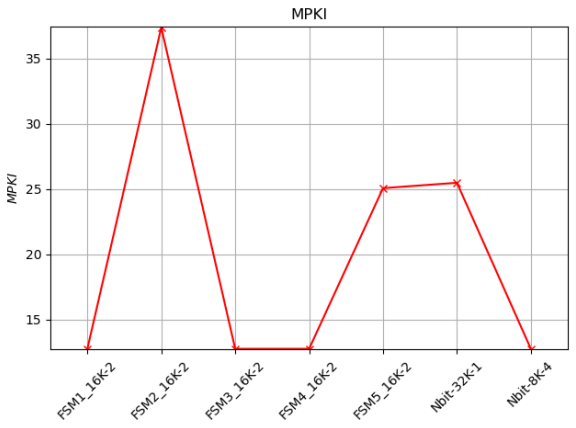


(d) 416.gamess

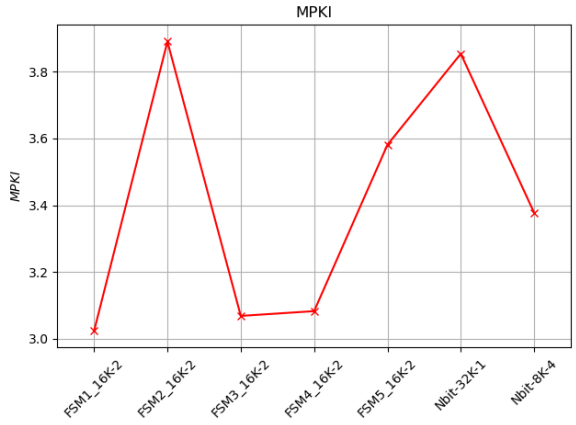
Figure 6: Branch Prediction Accuracy for Different 32K Predictors



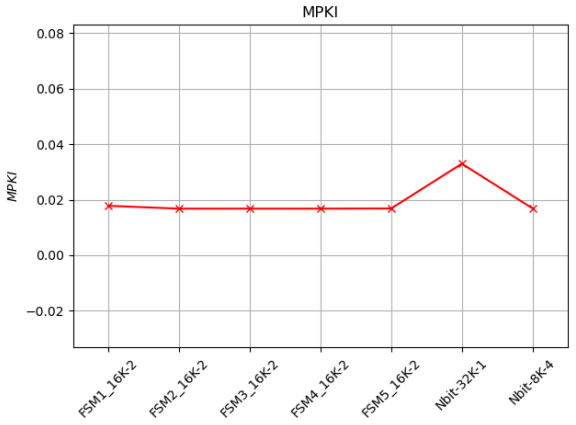
(e) 429.mcf



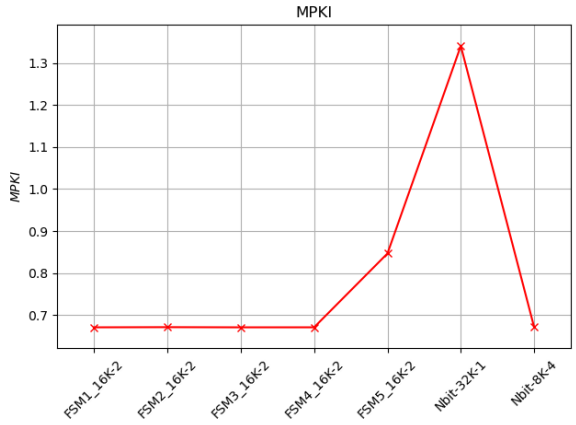
(f) 433.milc



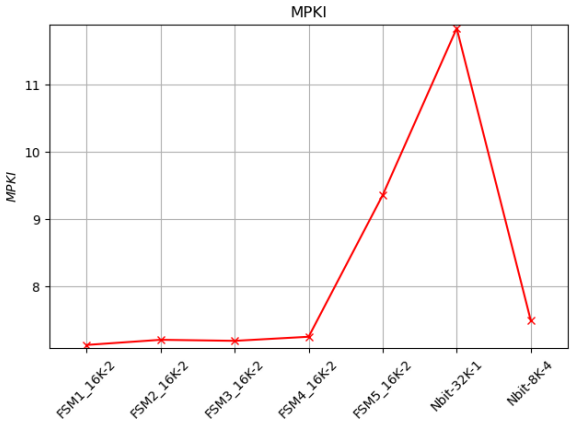
(g) 435.gromacs



(h) 436.cactusADM

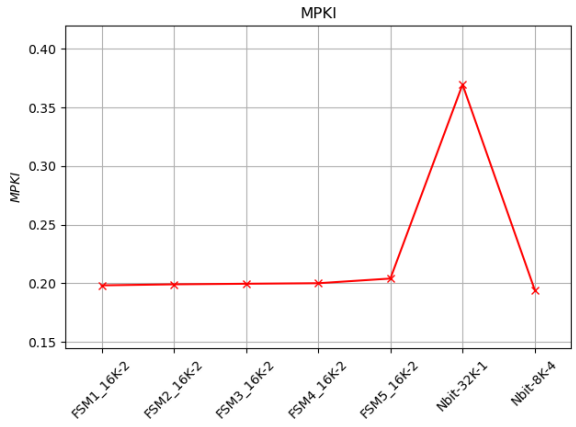


(i) 437.leslie3d

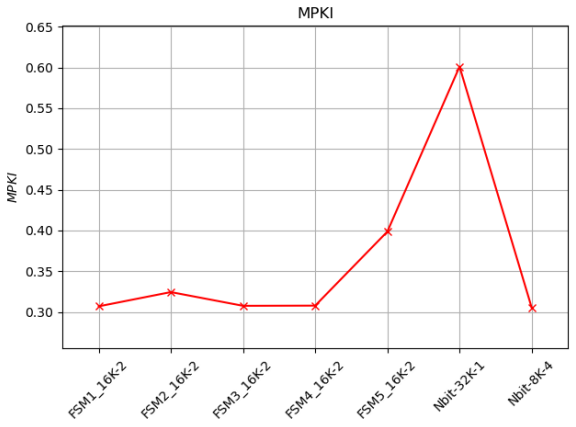


(j) 450.soplex

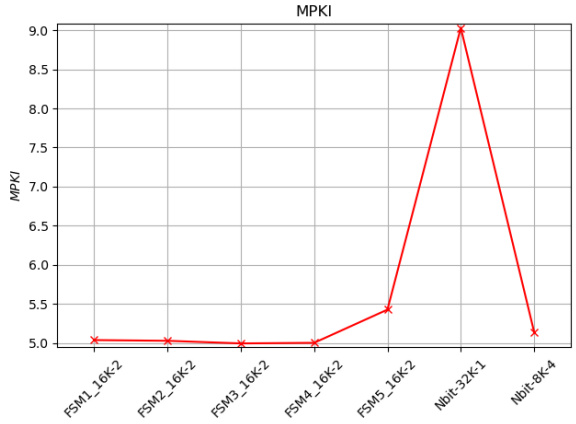
Figure 6: (continued)



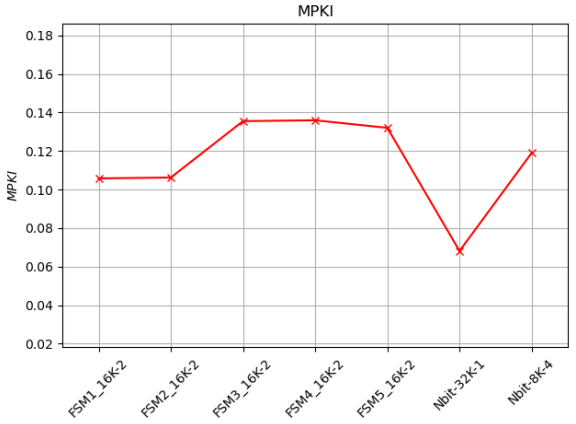
(k) 456.hmmer



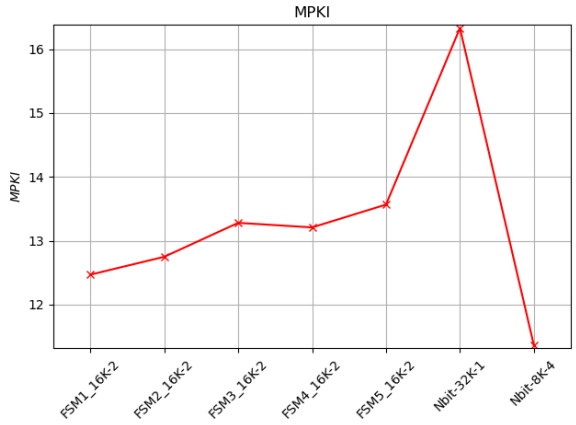
(l) 459.GemsFDTD



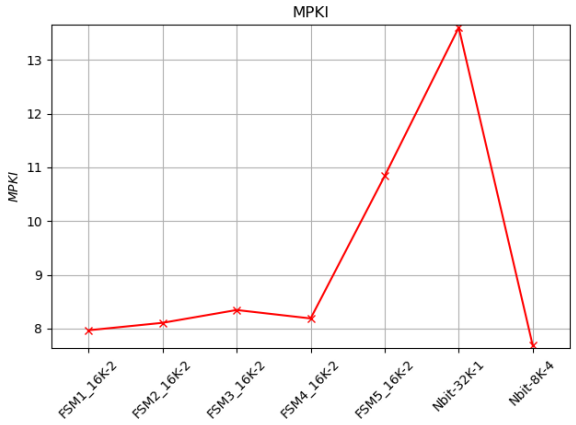
(m) 464.h264ref



(n) 470.lbm



(o) 471.omnetpp



(p) 483.xalancbmk

Figure 6: (continued)

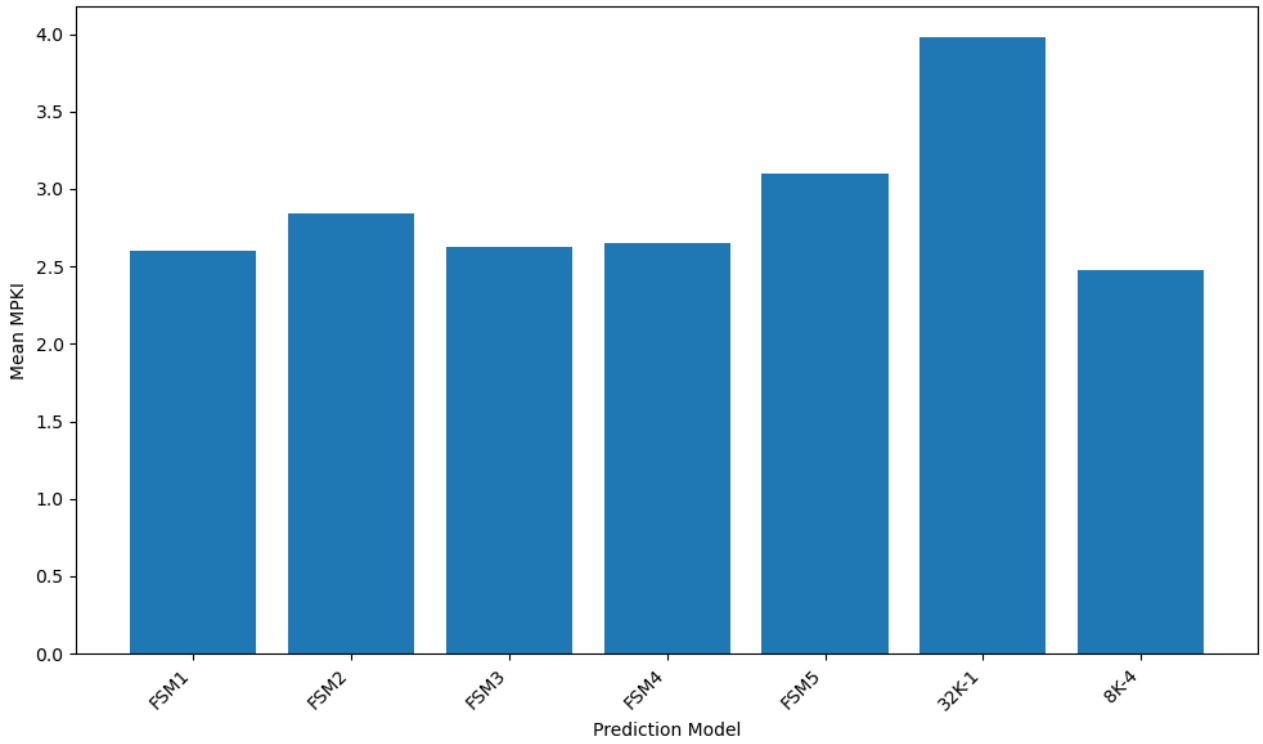


Figure 7: Geometric Mean MPKI for 32K predictors

Συνοψίζοντας, μεταξύ των εξεταζόμενων **Nbit predictors** με ίδιο συνολικό αποτύπωμα υλικού (**32K bits**), ο **Nbit-8K-4** παρουσιάζει τις **καλύτερες επιδόσεις** σε όρους MPKI, ξεπερνώντας ελαφρώς τον Nbit-16K-2. Επιπλέον, λόγω του μικρότερου αριθμού entries, προσφέρει και **μειωμένο latency**, γεγονός που τον καθιστά ταχύτερο στην πράξη. Αντιθέτως, ο Nbit-16K-2 εμφανίζει ελαφρώς **μικρότερη δυναμική κατανάλωση**, λόγω μικρότερου πλήθους toggled bits ανά update, όμως η διαφορά αυτή είναι περιορισμένη. Συνεπώς, για ένα σύστημα όπου προτεραιότητα έχουν οι επιδόσεις και η ταχύτητα, ο **Nbit-8K-4** αποτελεί την καταλληλότερη επιλογή.

4 Μελέτη του BTB

BTB Predictors

Οι BTB (Branch Target Buffer) predictors χρησιμοποιούνται για την πρόβλεψη του **διευθυνσιοδοτούμενου στόχου** (target address) ενός branch, σε περιπτώσεις όπου προβλέπεται ότι το branch θα εκτελεστεί (taken). Αντίθετα με τους κλασικούς branch predictors που απλώς αποφασίζουν αν το branch θα γίνει ή όχι, το BTB στοχεύει στο να επιταχύνει τη ροή εντολών προβλέποντας έγκαιρα **πού** θα μεταφερθεί ο έλεγχος.

Ένα BTB αποτελείται από έναν πίνακα εγγραφών (entries), όπου κάθε εγγραφή περιλαμβάνει:

- Τη διεύθυνση της branch εντολής (PC),
- Την προβλεπόμενη διεύθυνση στόχου (target address),
- Ενδεχομένως metadata όπως valid bits ή ιστορικά στοιχεία.

Κατά την προσπέλαση, το BTB αναζητά μια αντιστοιχία μεταξύ της τρέχουσας διεύθυνσης εντολής και των αποθηκευμένων PC. Αν βρεθεί τέτοια εγγραφή και η πρόβλεψη είναι taken, τότε το pipeline μπορεί να προφορτώσει το target address χωρίς καθυστέρηση, μειώνοντας σημαντικά τα **penalties από mispredicted branches**.

Θα εξετάσουμε τις εξής περιπτώσεις BTB predictors:

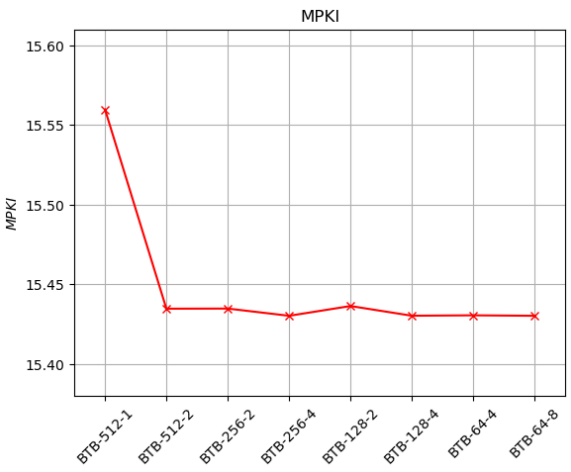
BTB Entries	BTB Associativity
512	1, 2
256	2, 4
128	2, 4
64	4, 8

Table 3: BTB Configurations (Entries and Associativity)

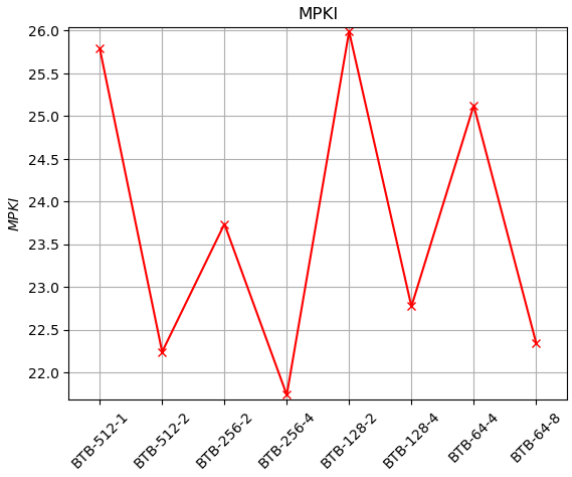
Η υλοποίηση παραλείπεται για λόγους συντομίας. Καλούμε τους παρακάτω BTB predictors στο cslab_branch.cpp:

```
VOID BTB()
{
    btb_predictors.push_back(new BTBPredictor(512, 1));
    btb_predictors.push_back(new BTBPredictor(512, 2));
    btb_predictors.push_back(new BTBPredictor(256, 2));
    btb_predictors.push_back(new BTBPredictor(256, 4));
    btb_predictors.push_back(new BTBPredictor(128, 2));
    btb_predictors.push_back(new BTBPredictor(128, 4));
    btb_predictors.push_back(new BTBPredictor(64, 4));
    btb_predictors.push_back(new BTBPredictor(64, 8));
}
```

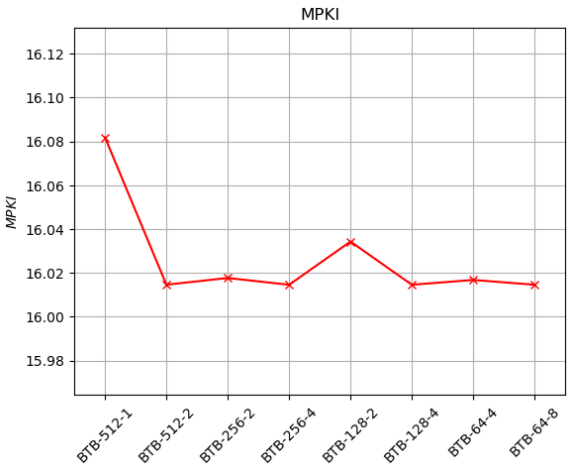
Και καταλήγουμε με τα εξής διαγράμματα:



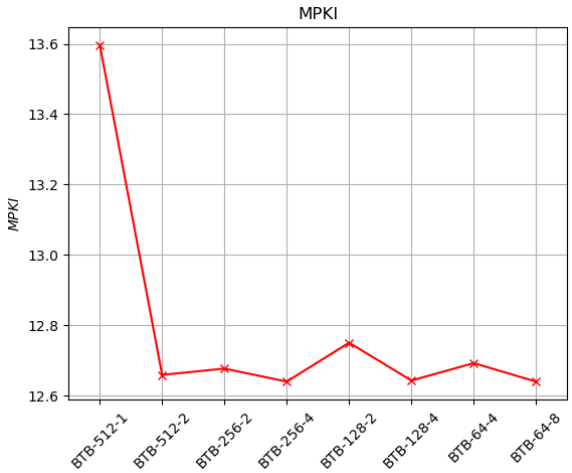
(a) 401.bzip2



(b) 403.gcc

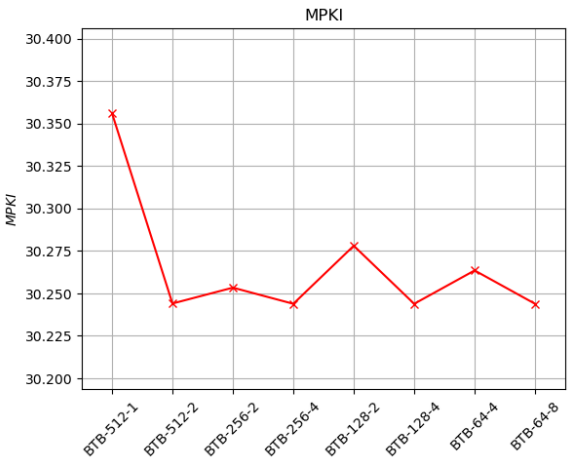


(c) 410.bwaves

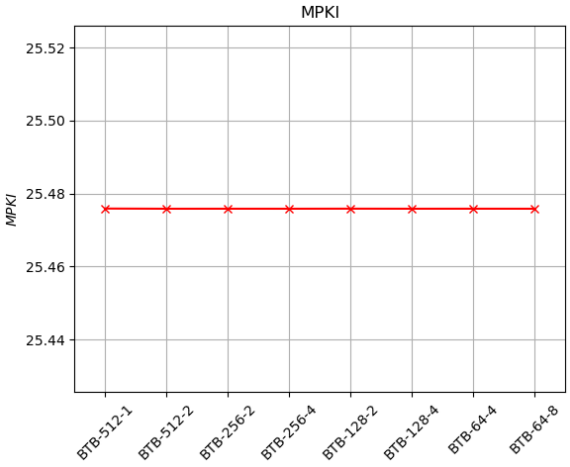


(d) 416.gamess

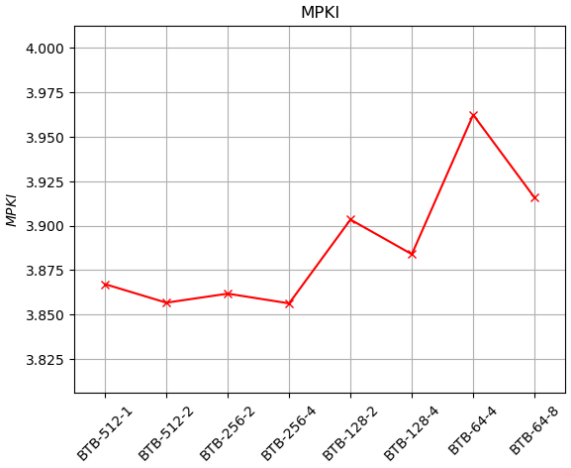
Figure 8: BTB Predictors Accuracy for Different Configurations



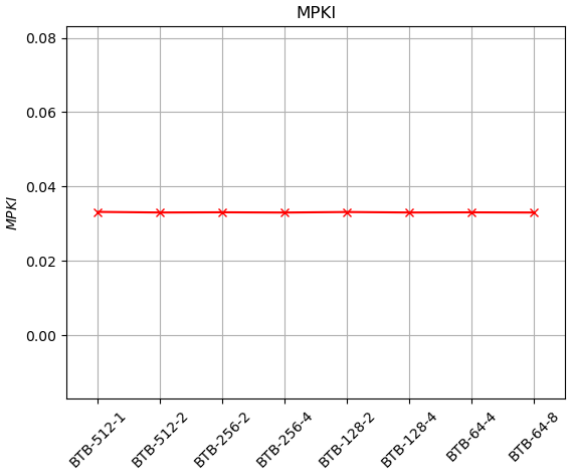
(e) 429.mcf



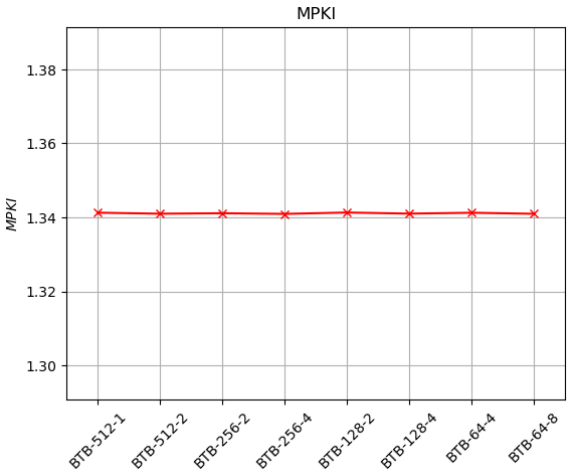
(f) 433.milc



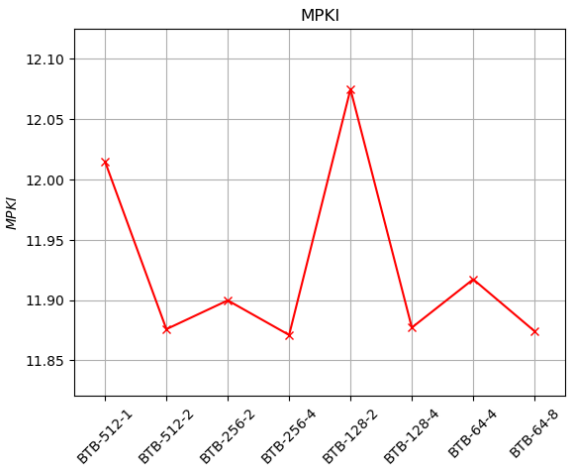
(g) 435.gromacs



(h) 436.cactusADM

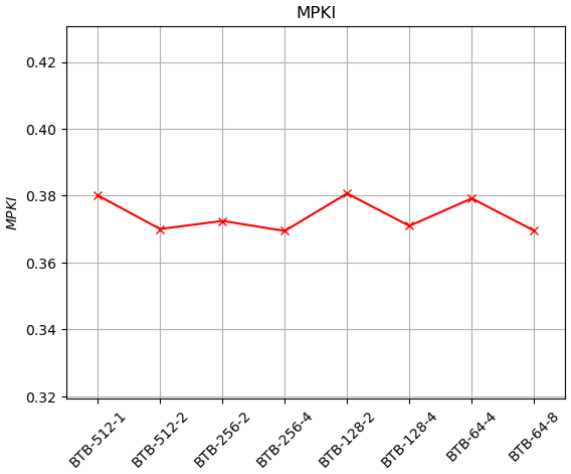


(i) 437.leslie3d

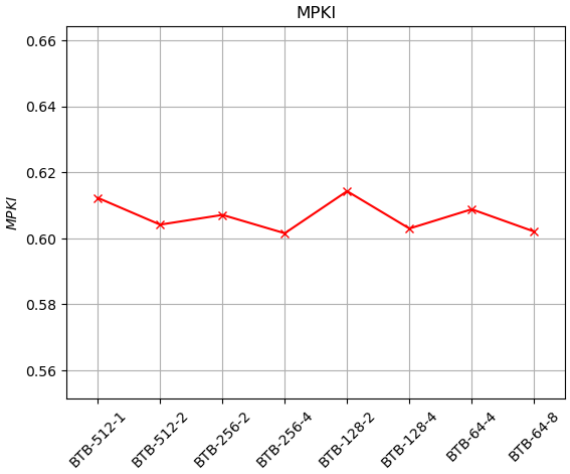


(j) 450.soplex

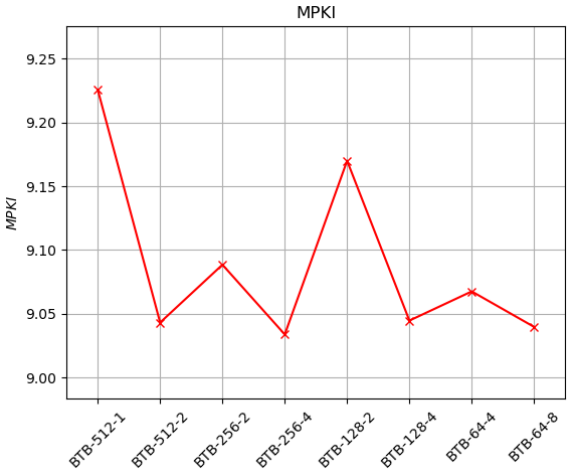
Figure 8: (continued)



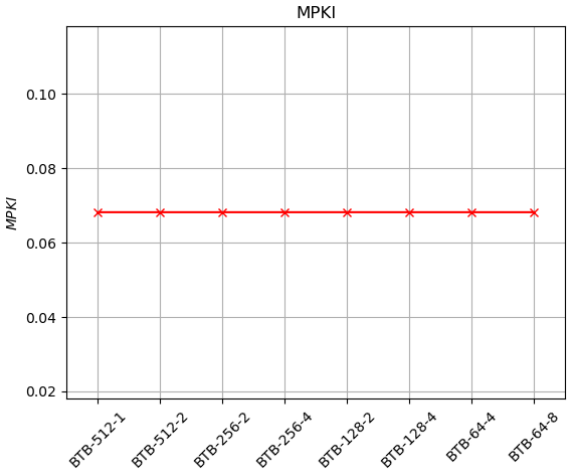
(k) 456.hmmer



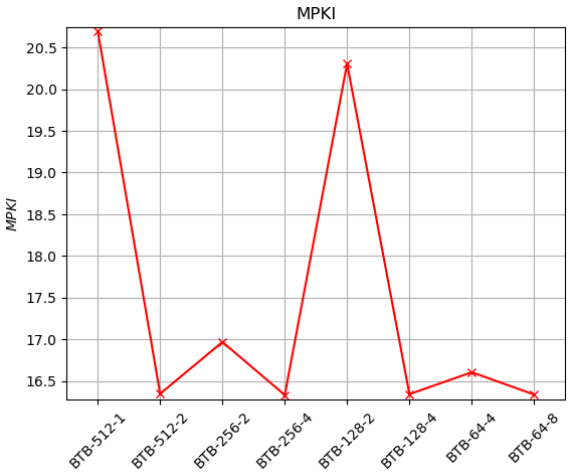
(l) 459.GemsFDTD



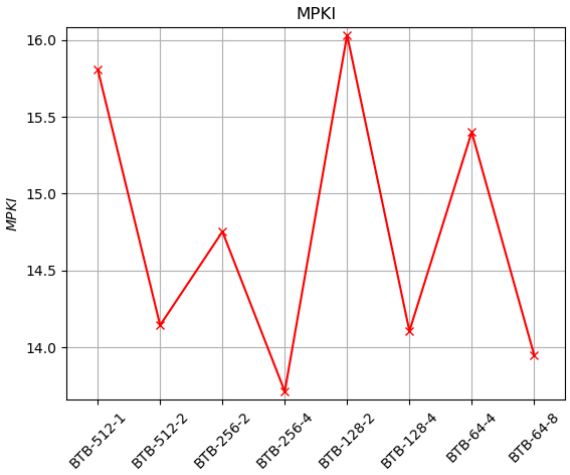
(m) 464.h264ref



(n) 470.lbm



(o) 471.omnetpp



(p) 483.xalancbmk

Figure 8: (continued)

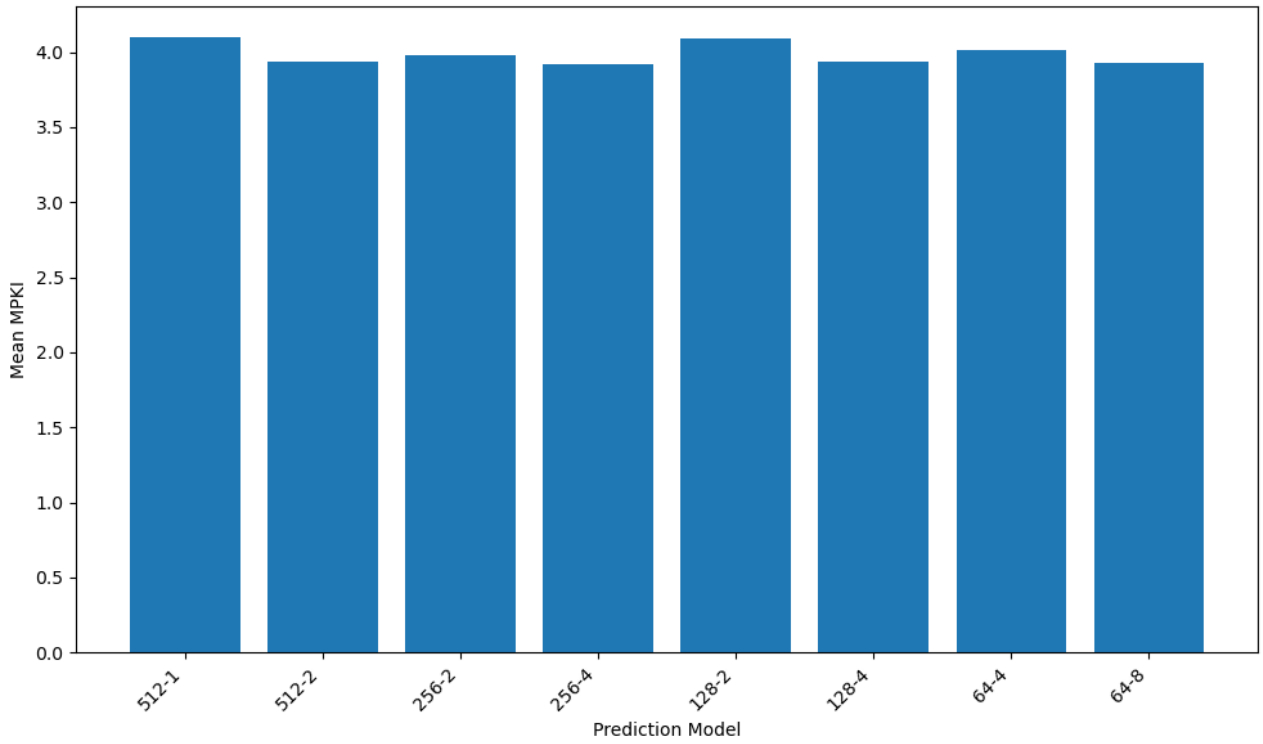


Figure 9: Geometric Mean MPKI for BTB predictors

Η απόδοση των διαφόρων διαμορφώσεων BTB επηρεάζεται από τον αριθμό των εγγραφών (entries) και την associativity, καθώς αυτά καθορίζουν την ικανότητα αποθήκευσης και γρήγορης ανάκτησης διευθύνσεων στόχων διακλαδώσεων.

Αριθμός Εγγραφών (Entries): Όσο περισσότερες οι εγγραφές, τόσο περισσότερες μοναδικές διακλαδώσεις μπορεί να αποθηκεύσει ο BTB, μειώνοντας τα *capacity misses*—δηλαδή, αστοχίες λόγω ανεπαρκούς χώρου.

Associativity: Ορίζει πόσες διαφορετικές διακλαδώσεις μπορούν να αντιστοιχηθούν στην ίδια ομάδα (set). Υψηλότερη associativity μειώνει τα *conflict misses*, δηλαδή αστοχίες που προκύπτουν από σύγκρουση πολλών διακλαδώσεων στον ίδιο περιορισμένο αριθμό θέσεων.

Επίδραση της Associativity (σταθερός αριθμός entries):

- 512-1 (4.10) → 512-2 (3.94): Μεγάλη βελτίωση από associativity 1 σε 2.
- 256-2 (3.98) → 256-4 (3.92): Μικρή αλλά μετρήσιμη βελτίωση.
- 128-2 (4.09) → 128-4 (3.94): Σημαντική βελτίωση.
- 64-4 (4.01) → 64-8 (3.93): Αξιοσημείωτη βελτίωση.

Επίδραση του Αριθμού Εγγραφών (σταθερή associativity):

- Associativity 2: 128-2 (4.09) → 256-2 (3.98) → 512-2 (3.94)
- Associativity 4: 64-4 (4.01) → 128-4 (3.94) → 256-4 (3.92)

Παρατηρήσεις:

- Οι καλύτερες επιδόσεις (χαμηλότερο MPKI) εμφανίζονται στις διαμορφώσεις: 256-4 (3.92), 64-8 (3.93), 512-2 (3.94), και 128-4 (3.94).
- Η αύξηση μόνο των entries χωρίς ανάλογη αύξηση στην associativity (π.χ., 512-1) οδηγεί σε χαμηλή απόδοση λόγω conflict misses.
- Η αύξηση της associativity αποδεικνύεται σταθερά ευεργετική για την απόδοση.

Επιλογή Διαμόρφωσης:
Η επιλογή του βέλτιστου BTB εξαρτάται από τον συμβιβασμό ανάμεσα στην απόδοση (χαμηλό MPKI) και το κόστος υλοποίησης (σε μέγεθος και πολυπλοκότητα). Ένας πρακτικός δείκτης μεγέθους είναι το γινόμενο *Entries* × *Associativity*:

- **256-4**: Καλύτερη απόδοση (3.92 MPKI), μέγεθος 256×4=1024.
- **64-8**: Ανταγωνιστική απόδοση (3.93 MPKI), μικρότερο μέγεθος 64×8=512.
- **128-4**: Εξίσου καλή απόδοση (3.94 MPKI), ίδιο μέγεθος με το 64-8.
- **512-2**: Καλή απόδοση (3.94 MPKI), αλλά μεγαλύτερο μέγεθος 512×2=1024.

Συμπέρασμα: Παρόλο που η διαμόρφωση **256-4** προσφέρει την απόλυτα καλύτερη απόδοση, οι διαμορφώσεις **64-8** και **128-4** αποτελούν εξαιρετικές εναλλακτικές με παρόμοια απόδοση και σημαντικά μικρότερο υπολογιζόμενο μέγεθος. Συνεπώς, για έναν καλό συμβιβασμό ανάμεσα σε απόδοση και κόστος υλικού, προτείνεται η διαμόρφωση **64-8** ή **128-4**.
Τελική Επιλογή: Το **BTB-256-4** αποτελεί την καλύτερη ισορροπία μεταξύ επίδοσης και κόστους υλοποίησης.

5 Μελέτη του RAS

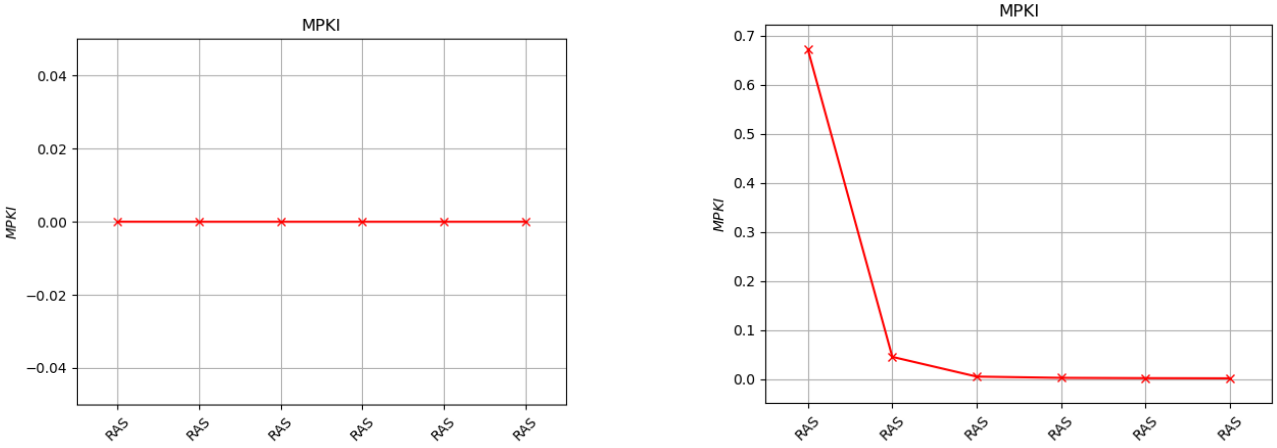
Η **Return Address Stack (RAS)** είναι ένας εξειδικευμένος μηχανισμός πρόβλεψης διακλαδώσεων που χρησιμοποιείται για την πρόβλεψη διευθύνσεων επιστροφής από υπορουτίνες (function calls). Κάθε φορά που εκτελείται μία εντολή `call`, η διεύθυνση επιστροφής αποθηκεύεται στη στοίβα της RAS. Όταν αργότερα εκτελείται μία `return`, η προβλεπόμενη διεύθυνση λαμβάνεται από την κορυφή της στοίβας. Αυτός ο μηχανισμός είναι εξαιρετικά αποτελεσματικός, καθώς οι `return` εντολές εμφανίζονται συχνά και η αλληλουχία `call-return` διατηρείται κατά κανόνα με μεγάλη ακρίβεια.

Η απόδοση της RAS επηρεάζεται κυρίως από το μέγεθός της, δηλαδή από τον αριθμό των διευθύνσεων που μπορεί να κρατήσει. Μια μικρή RAS μπορεί να "ξεχειλίσει" όταν υπάρχουν πολλές εμφωλευμένες κλήσεις (nested calls), οδηγώντας σε λάθος πρόβλεψη. Αντίθετα, μια πολύ μεγάλη RAS μπορεί να καταναλώνει περιττούς πόρους χωρίς σημαντική βελτίωση στην απόδοση.

Δοκιμασμένες διαμορφώσεις RAS:

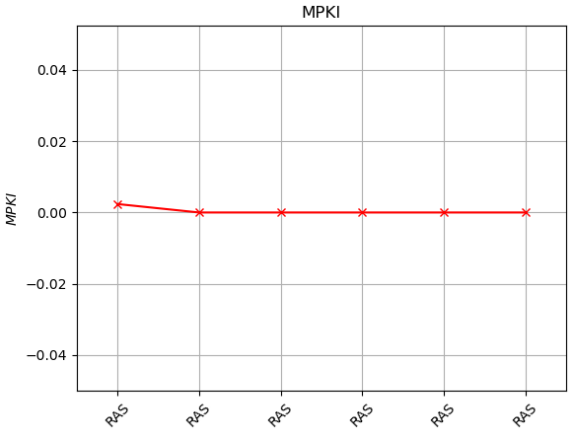
```
VOID InitRas()
{
    for (UINT32 i = 4; i <= 64; i*=2) {
        ras_vec.push_back(new RAS(i));
        if (i == 32)
            ras_vec.push_back(new RAS(48));

        }
}
```

Και καταλήγουμε με τα εξής διαγράμματα:

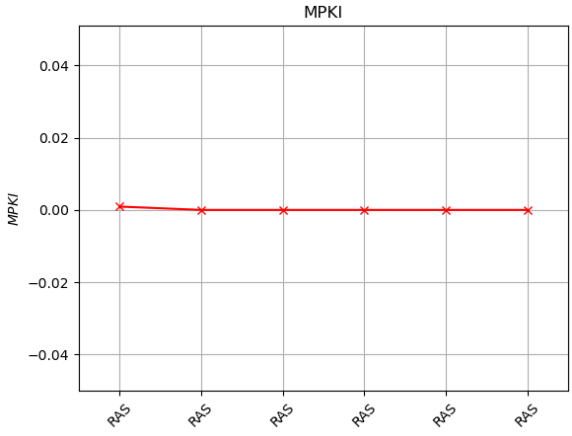


(a) 401.bzip2 (b) 403.gcc

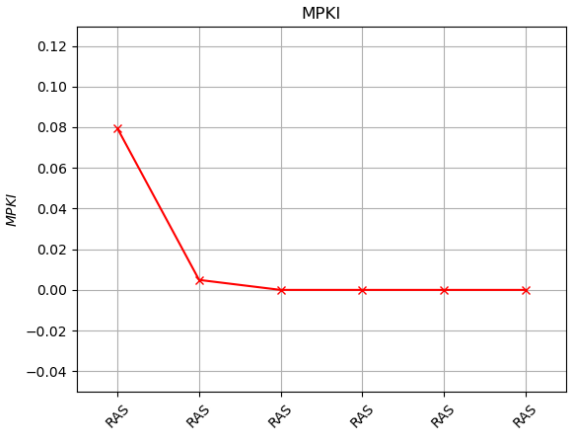
Figure 10: RAS Accuracy for Different Configurations



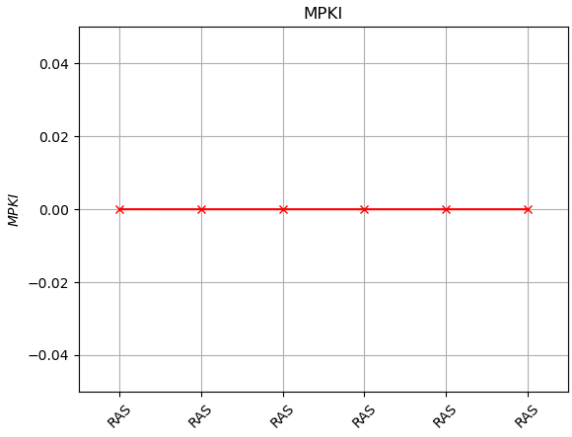
(c) 410.bwaves



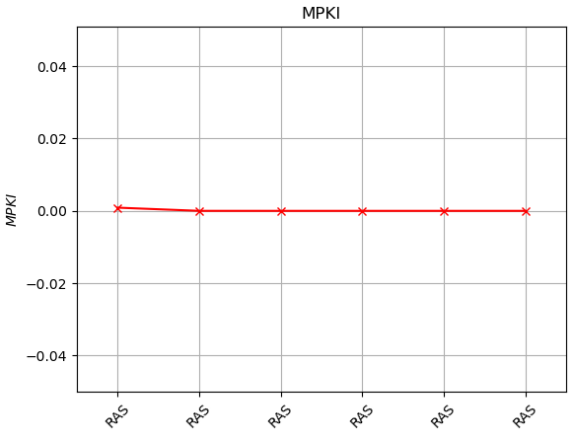
(d) 416.gamess



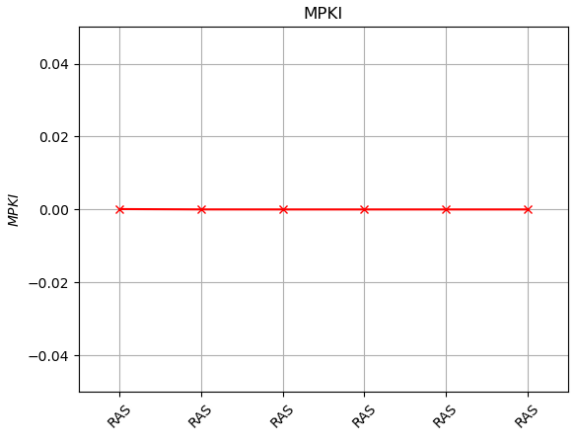
(e) 429.mcf



(f) 433.milc

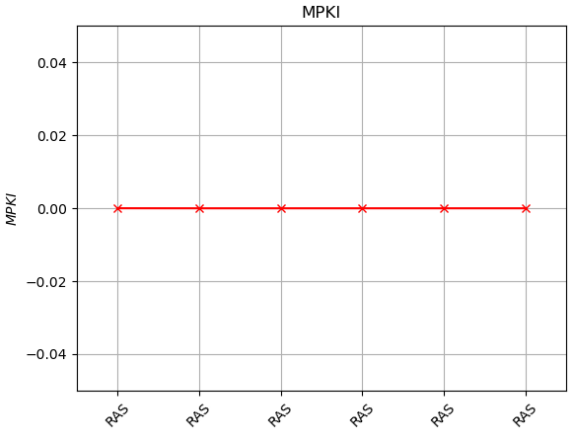


(g) 435.gromacs

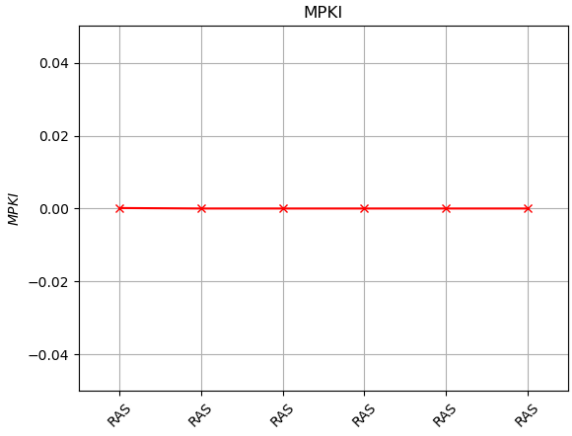


(h) 436.cactusADM

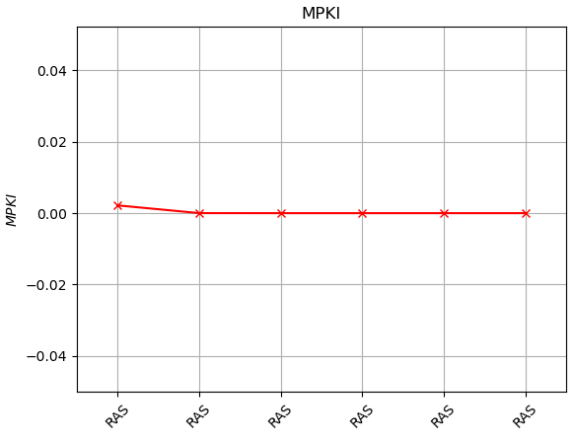
Figure 10: (continued)



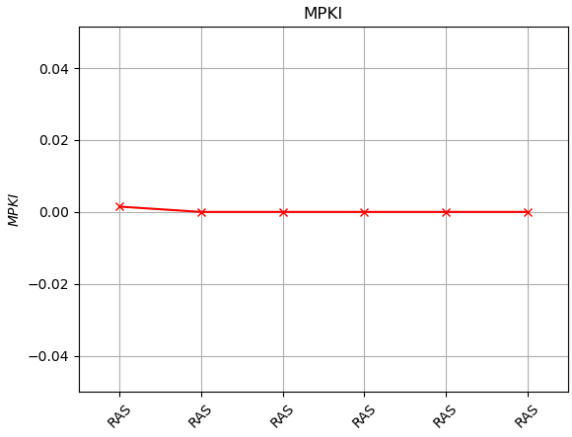
(i) 437.leslie3d



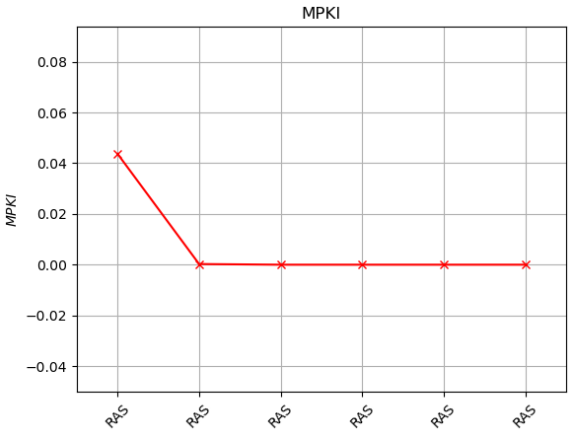
(j) 450.soplex



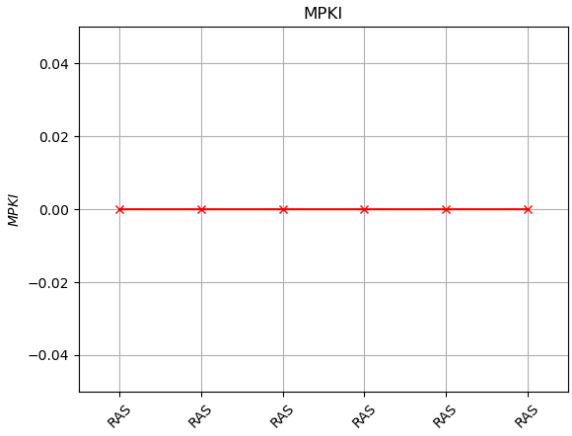
(k) 456.hmmer



(l) 459.GemsFDTD

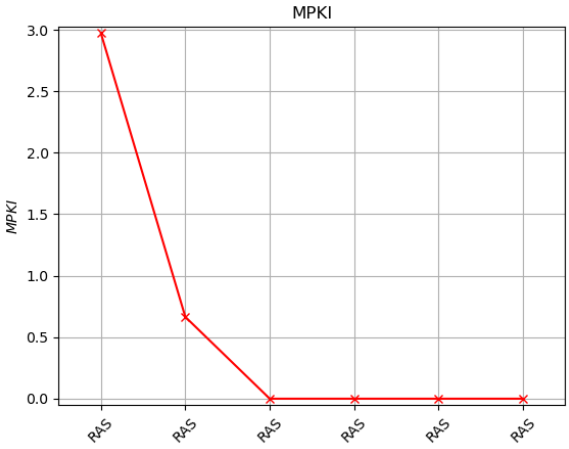


(m) 464.h264ref

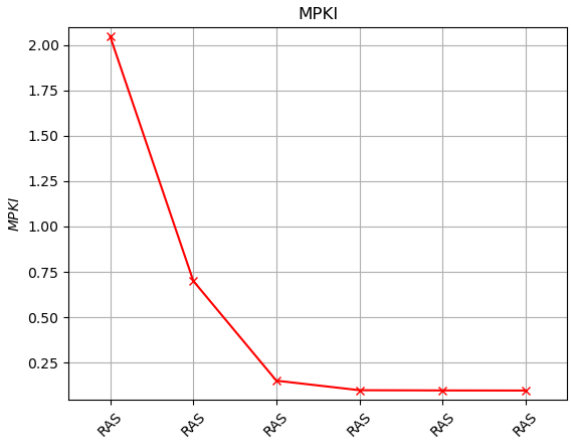


(n) 470.lbm

Figure 10: (continued)



(o) 471.omnetpp



(p) 483.xalancbmk

Figure 10: (continued)

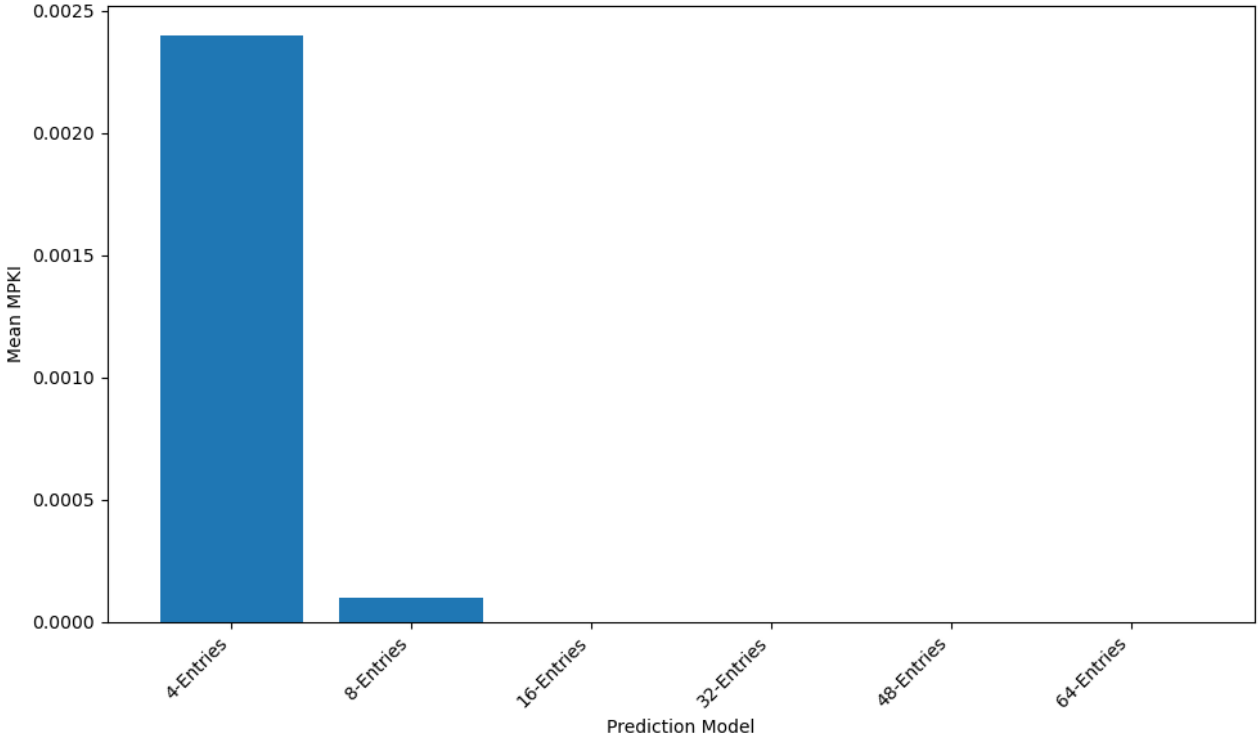


Figure 11: Geometric Mean MPKI for RAS predictors

Τα αποτελέσματα δείχνουν ότι όσο αυξάνεται το μέγεθος της RAS, η απόδοση βελτιώνεται σημαντικά, μέχρι το σημείο όπου οι λανθασμένες προβλέψεις πρακτικά εκμηδενίζονται.

- Με 4 εγγραφές, παρατηρείται MPKI = 0.0024.
- Με 8 εγγραφές, η απόδοση βελτιώνεται δραματικά (MPKI = 0.0001).
- Από τις 16 εγγραφές και πάνω, η RAS έχει μηδενικό MPKI.

Αυτό υποδηλώνει ότι για το σύνολο εφαρμογών που χρησιμοποιείται, 16 εγγραφές επαρκούν ώστε να καλύψουν το βάθος των εμφωλευμένων `call-return` ζευγών χωρίς να παρατηρείται υπερχείλιση της στοίβας.

Συμπέρασμα:
Η **RAS μεγέθους 16 εγγραφών** αποτελεί το ιδανικό σημείο ισορροπίας ανάμεσα σε απόδοση και κόστος υλοποίησης. Παρέχει **μηδενικές λανθασμένες προβλέψεις** και, συνεπώς, δεν υπάρχει πρακτικό όφελος από τη χρήση μεγαλύτερων διαμορφώσεων όπως 32, 48 ή 64 εγγραφές, καθώς δεν παρατηρείται επιπλέον βελτίωση. Από άποψης κόστους υλικού και ενεργειακής κατανάλωσης, η χρήση μιας περιττά μεγάλης RAS μπορεί να είναι αντιπαραγωγική.

6 Σύγκριση διαφορετικών predictors

Είδη Branch Predictors

- **Static Always Taken:** Ένας απλός στατικός predictor που υποθέτει πάντα ότι το branch θα γίνει (θα είναι taken). Παρότι είναι απλός και δεν απαιτεί σχεδόν καθόλου hardware, η ακρίβειά του εξαρτάται έντονα από το είδος του κώδικα – είναι καλός όταν η πλειοψηφία των branches είναι taken.
- **Static BTFNT (Backward Taken, Forward Not Taken):** Ένας πιο ”έξυπνος” στατικός predictor. Υποθέτει ότι τα backward branches (όπως οι βρόχοι) είναι taken και τα forward branches (όπως τα if-else) είναι not taken. Έχει μεγαλύτερη ακρίβεια από τον Always Taken σε πολλές περιπτώσεις, ειδικά σε βρόχους.
- **Local History Two-Level Predictors:** Οι τοπικοί history-based predictors χρησιμοποιούν έναν πίνακα ιστορικού (BHT) για κάθε branch, αποθηκεύοντας τη συμπεριφορά του συγκεκριμένου branch. Η ιστορία αυτή χρησιμοποιείται για την πρόσβαση σε έναν πίνακα προβλέψεων (PHT), βελτιώνοντας σημαντικά την ακρίβεια για branches με επαναλαμβανόμενα μοτίβα. Είναι πολύ αποτελεσματικοί για branches που εξαρτώνται από το δικό τους ιστορικό.
- **Global History Two-Level Predictors:** Αντί να αποθηκεύουν ιστορικό για κάθε branch ξεχωριστά, οι global predictors χρησιμοποιούν έναν κοινό καταχωρητή ιστορικού (Global History Register - BHR) που καταγράφει το μοτίβο εκτέλεσης όλων των πρόσφατων branches. Το global ιστορικό χρησιμοποιείται ως δείκτης στον πίνακα προβλέψεων. Είναι ιδιαίτερα χρήσιμοι για branches που εξαρτώνται από το αποτέλεσμα άλλων branches.
- **Alpha 21264 Predictor:** Ένας σύνθετος predictor που χρησιμοποιεί ταυτόχρονα local, global και choice predictors. Κάθε predictor δίνει τη δική του πρόβλεψη, και ένας meta-predictor επιλέγει ποια πρόβλεψη να εμπιστευθεί. Συνδυάζει τα πλεονεκτήματα των τοπικών και καθολικών predictors, προσφέροντας υψηλή ακρίβεια με σχετικά λογικό hardware overhead (29KB περίπου).
- **Tournament Hybrid Predictors:** Πρόκειται για γενικευμένη μορφή του Alpha 21264. Δύο ή περισσότεροι ανεξάρτητοι predictors (π.χ. ένας local και ένας global) λειτουργούν παράλληλα, και ένας meta-predictor αποφασίζει ποιον να εμπιστευθεί. Επιτρέπει μεγάλη ευελιξία και μπορεί να συνδυάσει οποιουσδήποτε τύπους predictors, π.χ. δύο global με διαφορετικά BHR μήκη. Προσφέρουν εξαιρετική ακρίβεια, ειδικά όταν οι συμμετέχοντες predictors εξειδικεύονται σε διαφορετικά μοτίβα branches.

Καλούμε τους εξής constructors:

```
branch_predictors.push_back(new AlwaysTakenPredictor());

// 2) BTFNT
branch_predictors.push_back(new BTFNTPredictor());

// 3) n-bit predictor
NbitPredictor *fourbitPred = new NbitPredictor(13, 4);
branch_predictors.push_back(fourbitPred);
// 4) Pentium-M
branch_predictors.push_back(new PentiumMBranchPredictor());

// 5, 6, 7) Local History Two Level
branch_predictors.push_back(new LocalHistoryPredictor(11, 8));
branch_predictors.push_back(new LocalHistoryPredictor(12, 4));
branch_predictors.push_back(new LocalHistoryPredictor(13, 2));

// 8, 9) Global History Two Level
branch_predictors.push_back(new GlobalHistoryPredictor(14, 2));
branch_predictors.push_back(new GlobalHistoryPredictor(13, 4));

// 10) Alpha21264
branch_predictors.push_back(new Alpha21264());

// 11, ..., 16) Tournament Hybrid Predictors
branch_predictors.push_back(new TournamentHybridPredictor(10,
```

```
    new NbitPredictor(13, 2), // 8K entries, 2bit predictor
    new NbitPredictor(12, 4)  // 4K entries, 4bit predictor
    ));
    branch_predictors.push_back(new TournamentHybridPredictor(11,
    new NbitPredictor(13, 2), // 8K entries, 2bit predictor
    new GlobalHistoryPredictor(13, 2) // 8K entries, 2bit global history predictor
    ));
    branch_predictors.push_back(new TournamentHybridPredictor(11,
    new NbitPredictor(13, 2), // 8K entries, 2bit predictor
    new LocalHistoryPredictor(12, 2, 12, 2) // local history predictor, BHT and PHT:4K entries 2bit
        each
    ));
    branch_predictors.push_back(new TournamentHybridPredictor(11,
    new LocalHistoryPredictor(12, 2, 12, 2),
    new GlobalHistoryPredictor(13, 2)
    ));
    branch_predictors.push_back(new TournamentHybridPredictor(11,
    new GlobalHistoryPredictor(13, 2),
    new GlobalHistoryPredictor(12, 4) // 4K entries, 4bit global history predictor
    ));
    branch_predictors.push_back(new TournamentHybridPredictor(11,
    new LocalHistoryPredictor(12, 2, 12, 2),
    new LocalHistoryPredictor(11, 4, 12, 2) // local history predictor,BHT:2K entries,4bit,PHT:4K
        entries,2bit
    ));
```

Για λόγους απλότητας, παραλείπονται οι υλοποιήσεις και τα αναλυτικά διαγράμματα των predictors. Παρατίθεται το τελικό διάγραμμα με τις επιδόσεις των predictors:

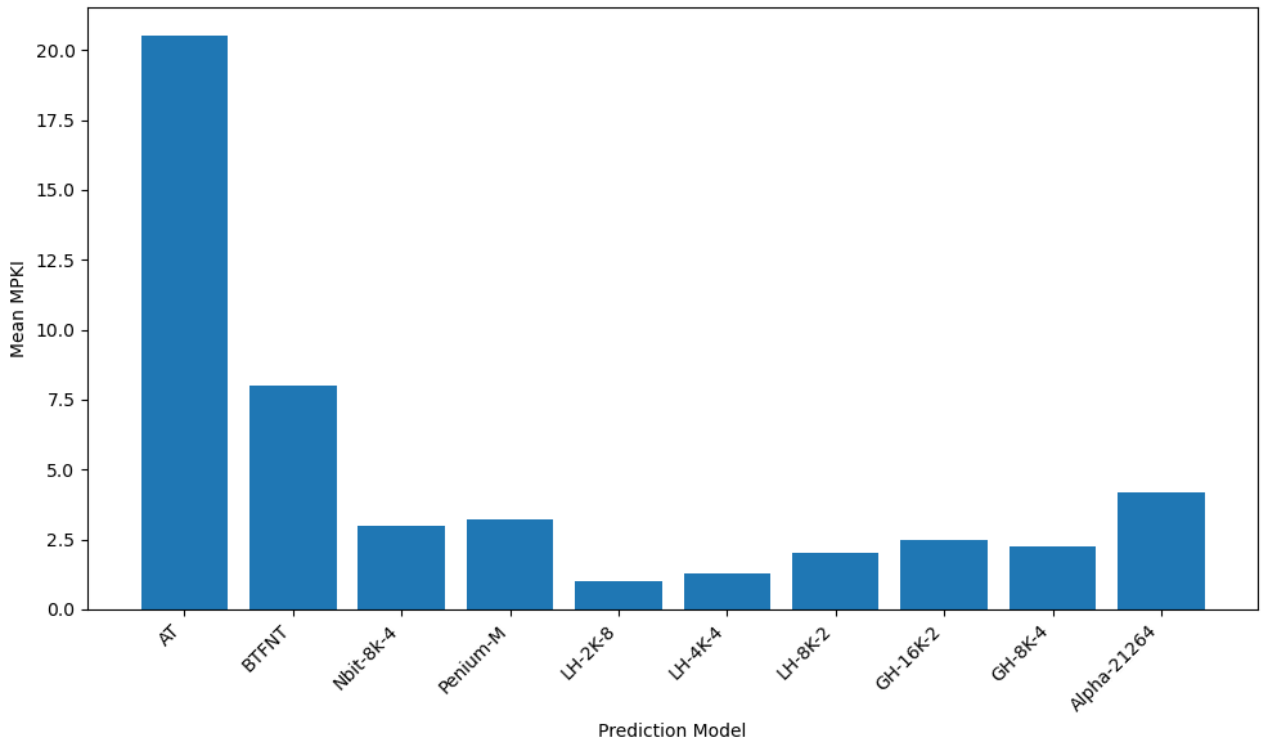


Figure 12: Geometric Mean MPKI for different predictors

Στον Πίνακα παρουσιάζονται τα αποτελέσματα της σύγκρισης των διαφόρων branch predictors.

Predictor	MPKI
Always Taken (AT)	20.5
Static BTFNT	8.0
Nbit-8K-4	3.0
Pentium-M	3.2
Local History 2K-8	**1.0**
Local History 4K-4	1.3
Local History 8K-2	2.0
Global History 16K-2	2.5
Global History 8K-4	2.25
Alpha 21264	4.2

Table 4: Σύγκριση απόδοσης διαφορετικών branch predictors (MPKI)

Από τα αποτελέσματα παρατηρούμε ότι οι **Local History two-level predictors** υπερτερούν των υπολοίπων, παρουσιάζοντας τις χαμηλότερες τιμές MPKI. Συγκεκριμένα, ο **Local History 2K-8** predictor παρουσιάζει την **καλύτερη επίδοση** με MPKI ίσο με 1.0.

Λαμβάνοντας υπόψη ότι όλοι οι predictors σχεδιάστηκαν ώστε να έχουν **παρόμοιο hardware overhead (32K)**, ο Local History 2K-8 αποτελεί την προτιμώμενη επιλογή, προσφέροντας την καλύτερη σχέση απόδοσης/κόστους.

Αν παρ' όλα αυτά η χρήση μεγάλου μήκους ιστορικού ($Z = 8$) θεωρηθεί μη επιθυμητή (π.χ. λόγω αυξημένης πολυπλοκότητας ή καθυστέρησης), ο **Local History 4K-4** αποτελεί μια εξαιρετική εναλλακτική με πολύ κοντινή απόδοση (MPKI = 1.3) και μικρότερο μήκος ιστορικού.

Συμπερασματικά, ο Local History 2K-8 αποτελεί την πλέον **αποδοτική και ισορροπημένη επιλογή** μεταξύ απόδοσης και υλοποιησιμότητας.

7 Ακρίβεια προσομοιώσεων

Στο σημείο αυτό, διερευνούμε κατά πόσο η χρήση των `train` inputs αντί των `ref` επηρεάζει τα συμπεράσματα που εξάγονται από τις προσομοιώσεις των branch predictors.

Χρησιμοποιούμε το ίδιο σύνολο από 16 predictors όπως στο Ερώτημα 5.6, το οποίο περιλαμβάνει τόσο στατικούς όσο και δυναμικούς predictors, καθώς και σύνθετους υβριδικούς predictors όπως ο Alpha 21264. Η σύγκριση εστιάζει κυρίως στο κατά πόσο η **σχετική κατάταξη των predictors** με βάση την απόδοσή τους (MPKI) παραμένει σταθερή όταν αλλάζει το input set.

Παρατίθεται το τελικό διάγραμμα με τις επιδόσεις των predictors:

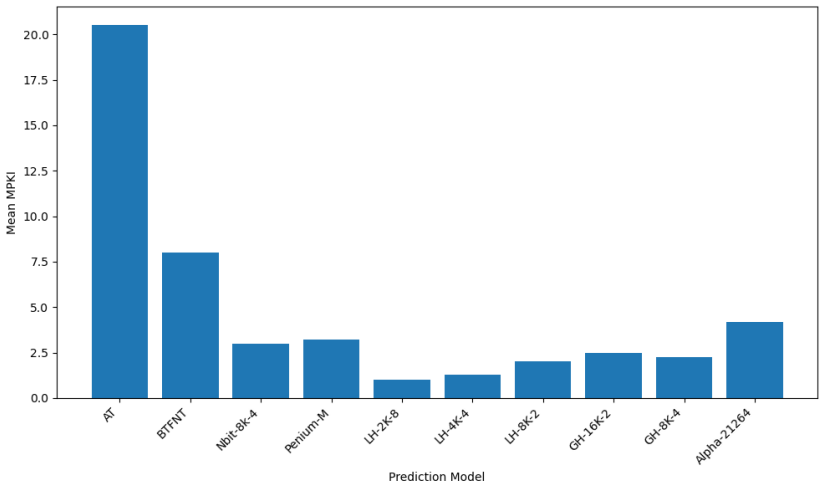


Figure 13: Geometric Mean MPKI for different predictors

Φαίνεται ότι η **σχετική κατάταξη των predictors** παραμένει σταθερή, με τον Local History 2K-8 να διατηρεί την καλύτερη απόδοση (MPKI = 1.0) και τους υπόλοιπους predictors να ακολουθούν σε παρόμοια σειρά.