Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Αναφορά για την 1η Σειρά Ασκήσεων

Μάθημα: Προηγμένα Θέματα Αρχιτεκτονικής Υπολογιστών, 8ο Εξάμηνο

Καθηγητές: Ν. Κοζύρης, Δ. Πνευματικάτος

Ονοματεπώνυμο: Γιάννης Πολυχρονόπουλος

Αριθμός Μητρώου: 03121089

Απρίλιος 27, 2025

Solution to Exercises

Γιάννης Πολυχρονόπουλος
AM: 03121089

Contents

Ανάλυση Εντολών Άλματος (Άσκηση 5.2)

1 - Σκοπός

Σκοπός του παρόντος πειράματος είναι η συλλογή στατιστικών σχετικά με τις εντολές άλματος που εκτελούνται από τα benchmarks, τόσο με train όσο και με ref inputs. Η κατηγοριοποίηση περιλαμβάνει τις εξής πέντε βασικές κατηγορίες:

- conditional taken,
- conditional not taken,
- unconditional,
- calls, and
- returns

2 - Μεθοδολογία

Για την καταγραφή των στατιστικών χρησιμοποιήθηκε το pintool **cslab_branch_stats.cpp** που δίνεται στο helpcode της άσκησης. Η συλλογή των αποτελεσμάτων έγινε αυτοματοποιημένα με τη χρήση python script, το οποίο παίρνει τα δεδομένα απο τα αρχεία εξόδου με τη μορφή:

```
Total Instructions: ...
Branch statistics:
  Total-Branches: ...
  Conditional-Taken-Branches: ...
  Conditional-NotTaken-Branches: ...
  ...
```

Για κάθε benchmark υπολογίστηκαν τα ποσοστά επί του συνόλου των εντολών άλματος.

3 - Αποτελέσματα (Train Inputs)

Οι παρακάτω πίνακες παρουσιάζουν το ποσοστό κάθε τύπου εντολής άλματος για κάθε benchmark χρησιμοποιώντας τα **train inputs** και τα **ref inputs** αντίστοιχα.

Table 1: Κατανομή Εντολών Άλματος για τα Train Inputs

Benchmark	Total Instructions	Total Branches	Cond Taken (%)	Cond Not Taken (%)	Unconditional (%)	Calls (%)	Returns (%)
401.bzip2	296379765992	48731209271	30.43	59.99	9.24	0.17	0.17
403.gcc	3184199944	754280993	35.75	40.41	8.83	7.50	7.50
410.bwaves	300165000496	31873090940	66.30	26.37	5.62	0.85	0.85
416.gamess	191137079973	14348666174	43.62	48.67	6.02	0.85	0.85
429.mcf	18199704025	3862678203	34.50	61.56	2.53	0.70	0.70
433.milc	38323372212	2240721101	55.43	24.15	0.96	9.73	9.73
435.gromacs	445004175352	18177452003	31.09	62.77	4.28	0.93	0.93
436.cactusADM	80776877586	170077639	83.63	11.03	0.98	2.18	2.18
437.leslie3d	215726997525	17218326673	95.25	4.71	0.03	0.01	0.01
450.soplex	7436065602	1255604054	50.71	38.46	3.63	3.60	3.60
456.hmmer	268784249864	13914142528	62.31	34.09	0.68	1.46	1.46
459.GemsFDTD	108729087090	3407688449	84.95	10.23	1.47	1.67	1.67
464.h264ref	495495984165	48075992539	50.49	17.66	6.92	12.46	12.46
470.lbm	96483397932	1356404964	13.01	67.99	18.61	0.19	0.19
471.omnetpp	210923279912	50445318207	21.70	41.35	9.04	13.96	13.96
483.xalancbmk	224077702969	54603312715	16.24	34.95	9.17	19.82	19.82

Table 2: Κατανομή Εντολών Άλματος για τα Ref Inputs

Benchmark	Total Instructions	Total Branches	Cond Taken (%)	Cond Not Taken (%)	Unconditional (%)	Calls (%)	Returns (%)
401.bzip2	288586987388	45686502043	34.86	49.52	9.75	2.93	2.93
403.gcc	142165130493	33846010305	36.33	40.1	9.26	7.15	7.15
410.bwaves	2181032745109	128568446784	68.91	26.46	3.0	0.82	0.82
416.gamess	1042736435082	73894107084	47.04	43.49	7.42	1.03	1.03
429.mcf	304114927152	67874045713	37.96	57.59	3.28	0.59	0.59
433.milc	1169067600886	67919982139	54.0	24.42	1.95	9.82	9.82
435.gromacs	2070986368394	84256708323	31.07	62.88	4.27	0.89	0.89
436.cactusADM	2782892990464	4364068375	95.22	3.73	0.19	0.43	0.43
437.leslie3d	1516975662927	119714174689	95.56	4.43	0.01	0.0	0.0
450.soplex	350312741178	57373565223	54.7	35.5	3.28	3.26	3.26
456.hmmer	875469100938	42158928449	64.99	33.74	0.53	0.38	0.38
459.GemsFDTD	1475062506601	47113489631	72.1	16.48	5.22	3.1	3.1
464.h264ref	495495984096	48075992508	50.49	17.66	6.92	12.46	12.46
470.lbm	1273858825997	15278517548	3.45	72.47	24.05	0.02	0.02
471.omnetpp	564555223493	138063849787	20.95	46.71	8.6	11.87	11.87
483.xalancbmk	921151284261	286420159340	26.42	56.5	2.61	7.24	7.24

Ο κώδικας που χρησιμοποιήθηκε για την δημιουργία των παραπάνω πινάκων είναι ο εξής:

Listing 1: Python script για parsing στατιστικών

```python
import os
import re
import csv

# === Folder setup ===
# input_dir = "/home/john/Adv_Comp_Arch/advcomparch-ex1-helpcode/outputs_stats/train"
input_dir = "/home/john/Adv_Comp_Arch/advcomparch-ex1-helpcode/outputs_stats/ref"
output_csv = "branch_stats_train.csv"

# === List with all the .out files ===
files = sorted([f for f in os.listdir(input_dir) if f.endswith(".out")])

# === Headers for CSV ===
header = [
    "Benchmark",
    "Total Instructions",
    "Total Branches",
    "Cond Taken (%)",
    "Cond Not Taken (%)",
    "Unconditional (%)",
    "Calls (%)",
    "Returns (%)"
]

rows = []

# === Function for parsing the file ===
def parse_file(filepath):
    with open(filepath, "r") as f:
        content = f.read()

    def extract_int(key):
        match = re.search(f"{key}:\\s+(\\d+)", content)
        if not match:
        print(f"[Warning] Field '{key}' not found at file: {filepath}")
            return 0
```

```
            return int(match.group(1))

        total_inst = extract_int("Total Instructions")
        total_br = extract_int("Total-Branches")
        ct = extract_int("Conditional-Taken-Branches")
        cnt = extract_int("Conditional-NotTaken-Branches")
        uncond = extract_int("Unconditional-Branches")
        calls = extract_int("Calls")
        rets = extract_int("Returns")

        # Percentages
        percent = lambda x: round(x / total_br * 100, 2) if total_br > 0 else 0

        return [total_inst, total_br, percent(ct), percent(cnt), percent(uncond), percent(calls), percent(rets)]

    # === Parse all the files ===
    for f in files:
        bench = f.split(".")[0]
        stats = parse_file(os.path.join(input_dir, f))
        rows.append([bench] + stats)

    # === Save the CSV ===
    with open(output_csv, "w", newline="") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(header)
        writer.writerows(rows)

    print(f"Saved at: {output_csv}")
```

4 - Παρατηρήσεις

Από την ανάλυση των στατιστικών των εντολών άλματος για τα *train και ref inputs*, όπως παρουσιάστηκαν στους Πίνακες 1 και 2, προκύπτουν οι ακόλουθες βασικές παρατηρήσεις:

- **Κοινά Στατιστικά:** Τα στατιστικά και στους δύο πίνακες είναι πολύ πανομοιότυπα. Αυτό δείχνει ότι η συμπεριφορά των benchmarks είναι σχετικά σταθερή και δεν επηρεάζεται σημαντικά από την είσοδο (input) που χρησιμοποιείται.

- **Κυριαρχία των Conditional Branches:** Στη συντριπτική πλειοψηφία των benchmarks, οι εντολές υπό συνθήκη (*conditional branches*) αποτελούν το μεγαλύτερο ποσοστό των συνολικών αλμάτων. Σε πολλά benchmarks, το ποσοστό αυτό ξεπερνά το 90%, όπως τα `436.cactusADM_ref` (99.99%) , τονίζοντας έτσι, την χρισιμότητα της ακριβούς πρόβλεψής τους για την απόδοση. Ακόμα και σε benchmarks με χαμηλότερο ποσοστό, όπως τα `464.h264ref` (68.2%) και `471.omnetpp` (67.7%), οι εντολές υπό συνθήκη παραμένουν η πολυπληθέστερη κατηγορία.

- **Μεγάλη Μεταβλητότητα στην Τάση των Conditional Branches:** Υπάρχει σημαντική διακύμανση στην αναλογία μεταξύ *conditional taken* και *conditional not taken* branches:

  - *Ισχυρή τάση προς Taken:* Benchmarks όπως τα `436.cactusADM` (95.2% taken), `437.leslie3d` (95.6% taken), `410.bwaves` (68.9% taken), `456.hmmer` (65.0% taken) και `459.GemsFDTD` (72.1% taken) δείχνουν σαφή προτίμηση προς την εκτέλεση του άλματος.
  - *Ισχυρή τάση προς Not Taken:* Αντίθετα, benchmarks όπως τα `470.lbm` (72.5% not taken), `435.gromacs` (62.9% not taken), `429.mcf` (57.6% not taken) και `483.xalancbmk` (56.5% not taken) δείχνουν προτίμηση στη μη εκτέλεση του άλματος.
  - *Πιο Ισορροπημένη Συμπεριφορά:* Άλλα benchmarks, όπως τα `416.gamess` με (47.0% taken / 43.5% not taken) και `403.gcc` (36.3% / 40.1%) παρουσιάζουν μια πιο ισορροπημένη κατανομή.

Αυτή η ποικιλομορφία καθιστά δύσκολη την επιτυχία απλών στατικών προγνωστών (static predictors) και αναδεικνύει την ανάγκη για δυναμικούς προγνώστες (dynamic predictors) που προσαρμόζονται στη συμπεριφορά του κάθε άλματος. Αυτό θα φανεί και σε benchmarks παρακάτω ερωτημάτων.

- **Σημασία των Calls/Returns για Ορισμένα Benchmarks:** Ένας αριθμός benchmarks εμφανίζει υψηλό ποσοστό εντολών κλήσης (*calls*) και επιστροφής (*returns*), που αθροιστικά ξεπερνούν το 10% ή και 20% των συνολικών αλμάτων. Χαρακτηριστικά παραδείγματα είναι τα `464.h264ref` (24.9%), `471.omnetpp` (23.7%), και `483.xalancbmk` (14.5%). Για αυτά τα benchmarks, η ακρίβεια της πρόβλεψης της διεύθυνσης επιστροφής μέσω της στοίβας διευθύνσεων επιστροφής (Return Address Stack - RAS) αναμένεται να έχει σημαντικό αντίκτυπο στην απόδοση. Αντίθετα, benchmarks όπως τα `437.leslie3d`, `470.lbm`, `436.cactusADM` και `456.hmmer` έχουν ελάχιστες τέτοιες εντολές (1%).

5 - Συμπεράσματα

- Παρατηρούμε ότι στα περισσότερα benchmarks κυριαρχούν οι *conditional branches*, με το ποσοστό των *taken* και *not taken* να παρουσιάζει σημαντική μεταβλητότητα ανάλογα με τη φύση της εφαρμογής.
- Τα benchmarks με έντονη χρήση υπορουτινών (π.χ. `403.gcc`) εμφανίζουν υψηλό ποσοστό σε *calls* και *returns*, γεγονός που θα επηρεάσει την ακρίβεια του RAS στα επόμενα ερωτήματα.
- Η αναλογία μεταξύ *conditional* και *unconditional* branches μπορεί να καθορίσει την αποτελεσματικότητα των διαφόρων branch predictors που θα εξεταστούν στις ασκήσεις 5.3 και 5.6.

N-Bit Predictors(Άσκηση 5.3)

1 - N = 1, 2, 3, 4 με 16K Entries (Ερώτημα i)

Στο πρώτο αυτό ερώτημα, μελετάται η απόδοση των N-bit κορεσμένων μετρητών (saturating counters). Διατηρήθηκε σταθερός ο αριθμός των καταχωρήσεων στον πίνακα ιστορικού διακλαδώσεων (Branch History Table - BHT) σε $2^{14} = 16384$ (περίπου 16K entries). Προσομοιώθηκαν οι περιπτώσεις για $N = 1, 2, 3,$ και 4 bits ανά καταχώρηση, χρησιμοποιώντας τα **ref inputs** για όλα τα benchmarks.

Αλλάξαμε το `run_train_predictors.sh` σε `run_ref_predictors.sh` και τρέξαμε το script για τα benchmarks με τα **ref inputs**. Η αρχικοποίηση των predictors έγινε στο αρχείο `cslab_branch.cpp` μέσω της συνάρτησης `InitPredictors()`, όπως φαίνεται στο παρακάτω κώδικα 2.

Listing 2: Αρχικοποίηση N-bit predictors (16K entries, N=1..4)

```
VOID InitPredictors(){
    for (int i=1; i <= 4; i++) {
        NbitPredictor *nbitPred = new NbitPredictor(14, i); // 2^14 ~= 16k
        branch_predictors.push_back(nbitPred);
    }
}
```

Για κάθε .out αρχείο που παράγεται από το pintool, τρέχουμε το δωσμένο python script (`plot_mpki_ipc.py`), το οποίο παράγει τα γραφήματα που απεικονίζουν την απόδοση των N-bit predictors. Το script αυτό διαβάζει τα αρχεία εξόδου και δημιουργεί γραφήματα με την απόδοση (MPKI) των predictors.

Στο παρακάτω σχήμα 1 απεικονίζεται η απόδοση των N-bit predictors (N=1, 2, 3, 4) με 16K BHT entries για τα **ref inputs**.

1.1 - Συμπεράσματα και Επιλογή Predictor (N=1..4, 16K Entries):

Η ανάλυση της απόδοσης των N-bit predictors, μετρούμενης σε MPKI, οδηγεί στα ακόλουθα συμπεράσματα. Υπενθυμίζεται ότι χαμηλότερη τιμή MPKI αντιστοιχεί σε καλύτερη απόδοση του predictor.

Γενικά, παρατηρείται βελτίωση της απόδοσης (μείωση του MPKI) καθώς αυξάνεται ο αριθμός των bits N ανά καταχώρηση στο BHT. Αυτό είναι αναμενόμενο, καθώς οι μετρητές με περισσότερα bits επιτρέπουν την αποθήκευση περισσότερης ιστορικότητας για τη συμπεριφορά της διακλάδωσης, οδηγώντας σε ακριβέστερες προβλέψεις.

Ωστόσο, η βελτίωση αυτή παρουσιάζει φθίνουσα απόδοση. Η μετάβαση από $N = 1$ σε $N = 2$ προσφέρει, στα περισσότερα benchmarks, τη σημαντικότερη μείωση του MPKI. Αντίθετα, η περαιτέρω αύξηση του N από 2 σε 3, και από 3 σε 4, οδηγεί σε πολύ μικρότερες, συχνά οριακές, βελτιώσεις στην ακρίβεια πρόβλεψης.

Λαμβάνοντας υπόψη ότι:

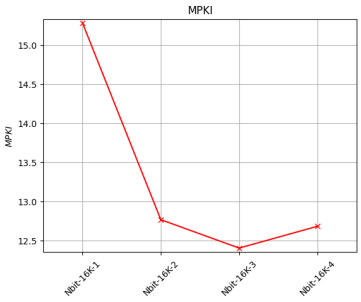
1. Η αύξηση του N συνεπάγεται αύξηση του απαιτούμενου hardware (περισσότερα bits αποθήκευσης ανά καταχώρηση στο BHT) και ενδεχομένως της πολυπλοκότητας του predictor.

2. Η σημαντικότερη βελτίωση στην ακρίβεια επιτυγχάνεται κατά τη μετάβαση από $N = 1$ σε $N = 2$.

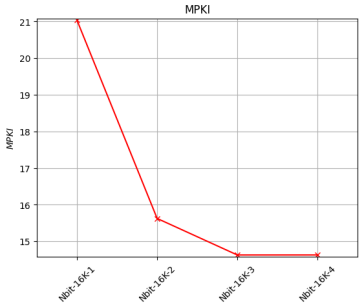
3. Οι μεταβάσεις $N = 2 \rightarrow 3$ και $N = 3 \rightarrow 4$ προσφέρουν οριακά οφέλη στην πλειονότητα των περιπτώσεων.

καταλήγουμε ότι ο **2-bit predictor (N=2)** αποτελεί την καλύτερη επιλογή υπό αυτές τις συνθήκες (σταθερός αριθμός 16K entries). Προσφέρει μια εξαιρετική ισορροπία μεταξύ της ακρίβειας πρόβλεψης και του κόστους υλικού, αξιοποιώντας τα οφέλη της προσθήκης ιστορικότητας χωρίς το δυσανάλογο κόστος των predictors με περισσότερα bits.

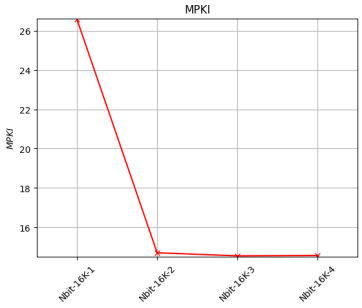
Figure 1: Απόδοση (MPKI) N-bit predictors (N=1, 2, 3, 4) με 16K BHT entries



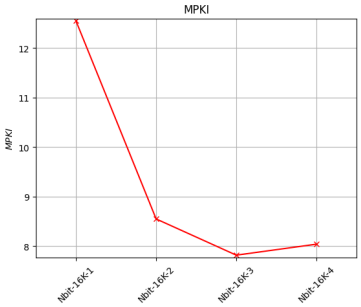
(a) 401.bzip2



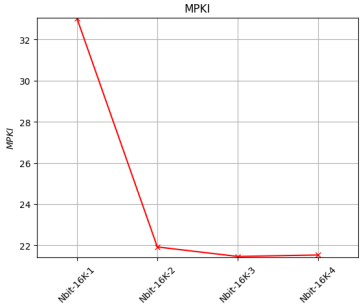
(b) 403.gcc



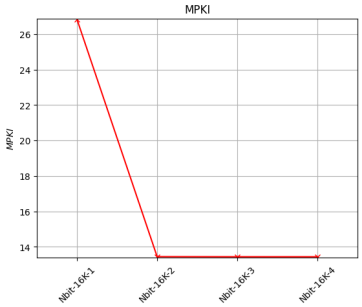
(c) 410.bwaves



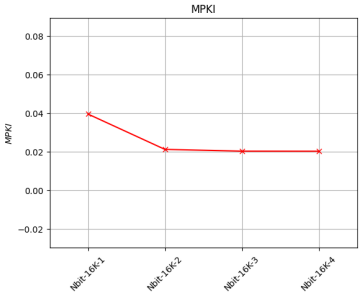
(d) 416.gamess



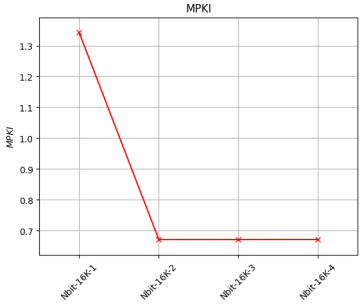
(e) 429.mcf



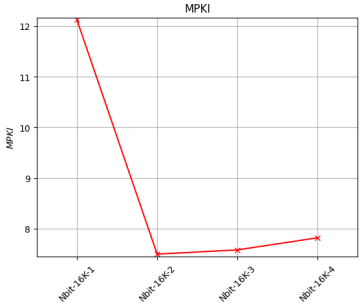
(f) 433.milc



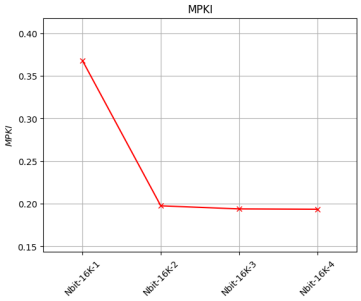
(g) 436.cactusADM



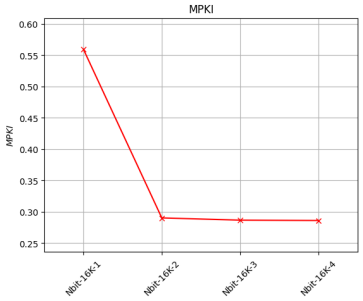
(h) 437.leslie3d



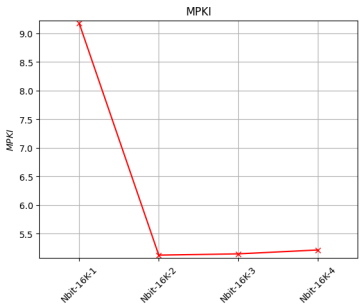
(i) 450.soplex



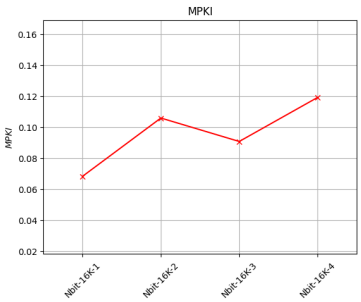
(j) 456.hmmer



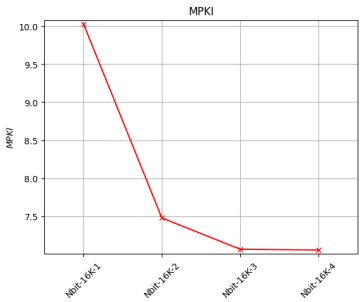
(k) 459.GemsFDTD



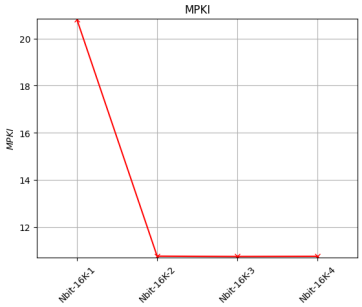
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

2 - Εναλλακτικά FSMs για N = 2 (Ερώτημα ii)

Στο δεύτερο αυτό ερώτημα, μελετήθηκε η απόδοση εναλλακτικών μηχανών πεπερασμένων καταστάσεων (Finite State Machines - FSMs) για 2-bit predictors ($N = 2$), όπως αυτές ορίζονται στον Πίνακα VI της δημοσίευσης Optimal 2-Bit Branch Predictors. Συγκεκριμένα, προσομοιώθηκαν τα FSMs που αντιστοιχούν στις γραμμές 1 έως 5 του πίνακα, διατηρώντας τον αριθμό των καταχωρήσεων του BHT σταθερό σε $2^{14} = 16384$ (16K entries), όπως και στο προηγούμενο ερώτημα. Ο predictor της γραμμής 1 (Row 1), είναι ο standard 2-bit κορεσμένος μετρητής (saturating counter) που εξετάστηκε και στο Ερώτημα (i).

Η αρχικοποίηση των predictors στο `cslab_branch.cpp` περιλαμβάνει τόσο τον standard N-bit predictor όσο και τους νέους N-bit Predictors με τα αντίστοιχα FSM, όπως φαίνεται στο κώδικα 3.

Listing 3: Αρχικοποίηση N-bit predictors (N=2, 16K entries)

```
VOID InitPredictors()
{
    /* Question 5.3 (ii) */
    // Row 1
    NbitPredictor *nbitPred = new NbitPredictor(14, 2); // 2-bit saturating counter
    branch_predictors.push_back(nbitPred);
    for (int row = 2; row <= 5; row++)
    {
        branch_predictors.push_back(new FSMPredictor(row));
    }
}
```

Η υλοποίηση των FSM predictors (Rows 2-5) βασίστηκε στην κλάση `FSMPredictor`, η οποία κληρονομεί από την `BranchPredictor`. Η κλάση αυτή χρησιμοποιεί έναν πίνακα μεταβάσεων (`transitions`) για να καθορίσει την επόμενη κατάσταση του 2-bit μετρητή, βάσει της τρέχουσας κατάστασης και του αποτελέσματος της διακλάδωσης (Taken/Not Taken). Ο πίνακας αυτός κωδικοποιεί τη λογική των διαφορετικών FSMs που παρουσιάζονται στο πίνακα του paper. Ο κώδικας της κλάσης και ο πίνακας μεταβάσεων παρατίθενται στο παρακάτω κώδικα 4.

Listing 4: Αρχικοποίηση FSM predictor Class(2-5)

```
class FSMPredictor : public BranchPredictor
{
public:
    FSMPredictor(unsigned row_) : BranchPredictor(), row(row_), index_bits(14), cntr_bits(2)
    {
        if (row < 2 || row > 5)
        {
            std::cerr << "Error: FSMPredictor row must be between 2 and 5." << std::endl;
            exit(1);
        }
        table_entries = 1 << index_bits;
        TABLE = new uint8_t[table_entries];
        memset(TABLE, 0, table_entries * sizeof(*TABLE));
    }

    ~FSMPredictor()
    {
        delete[] TABLE;
    }

    bool predict(ADDRINT ip, ADDRINT target) override
    {
        unsigned int ip_table_index = ip % table_entries;
        uint8_t ip_table_value = TABLE[ip_table_index];
        uint8_t prediction = ip_table_value >> (cntr_bits - 1);
        return (prediction != 0);
    }

    void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target) override
    {
        unsigned int idx = ip % table_entries;
        uint8_t state = TABLE[idx];

        unsigned int row_idx = row - 2;
        unsigned int outcome_idx = actual ? 1 : 0;
        uint8_t next_state = transitions[row_idx][outcome_idx][state];

        // Update the state in the table
        TABLE[idx] = next_state;

```

```
        // Update the counters
        updateCounters(predicted, actual);
    }

    std::string getName() override
    {
        std::ostringstream stream;
        stream << "FSM-Row-" << row;
        return stream.str();
    }

private:
    static const uint8_t transitions[4][2][4];
    unsigned int row;
    unsigned int table_entries;
    const unsigned index_bits, cntr_bits;
    uint8_t *TABLE;
};
const uint8_t FSMPredictor::transitions[4][2][4] = {
    // Row 2 (row_idx=0) - [outcome][state]
    {{0, 0, 0, 2}, // Not Taken transitions (state 0->0, 1->0, 2->0, 3->2)
     {1, 2, 3, 3}}, // Taken transitions (state 0->1, 1->2, 2->3, 3->3)
    // Row 3 (row_idx=1) - [outcome][state]
    {{0, 0, 1, 2}, // Not Taken transitions (state 0->0, 1->0, 2->1, 3->2)
     {1, 3, 3, 3}}, // Taken transitions (state 0->1, 1->3, 2->3, 3->3)
    // Row 4 (row_idx=2) - [outcome][state]
    {{0, 0, 0, 2}, // Not Taken transitions (state 0->0, 1->0, 2->0, 3->2)
     {1, 3, 3, 3}}, // Taken transitions (state 0->1, 1->3, 2->3, 3->3)
    // Row 5 (row_idx=3) - [outcome][state]
    {{0, 0, 1, 2}, // Not Taken transitions (state 0->0, 1->0, 2->1, 3->2)
     {1, 3, 3, 2}} // Taken transitions (state 0->1, 1->3, 2->3, 3->2)
};
```

2.1 - Συμπεράσματα και Επιλογή FSM (N=2, 16K Entries):

Η σύγκριση της απόδοσης (MPKI) των benchmarks (Figure 3) των εναλλακτικών 2-bit FSMs (Rows 1-5) στα 16K entries οδηγεί στις παρακάτω παρατηρήσεις:

- **Υπεροχή του Standard Saturating Counter (Row 1):** Στην πλειονότητα των benchmarks, ο standard 2-bit κορεσμένος μετρητής (FSM-Row-1) επιδεικνύει την καλύτερη ή μία από τις καλύτερες επιδόσεις (χαμηλότερο MPKI) μεταξύ των εξεταζόμενων 2-bit σχημάτων.
- **Απόδοση Εναλλακτικών FSMs:**
  - Τα FSMs των Rows 3 και 4 παρουσιάζουν συχνά παρόμοια απόδοση μεταξύ τους, η οποία είναι ελαφρώς χειρότερη από αυτή του Row 1 στα περισσότερα benchmarks.
  - Το FSM της Row 5 εμφανίζει σταθερά τη χειρότερη απόδοση από όλα τα εξεταζόμενα σχήματα.

Ανάλυση Συμπεριφοράς και Επιλογή Predictor:

Η διαφοροποίηση στην απόδοση πηγάζει από τις διαφορετικές στρατηγικές μετάβασης μεταξύ των καταστάσεων (Strongly Not Taken - SN, Weakly Not Taken - WN, Weakly Taken - WT, Strongly Taken - ST) που υιοθετεί κάθε FSM (2).

- **FSM 1 (Standard Saturating Counter - Διάγραμμα 1):** Παρατηρείται πως απαιτούνται δύο διαδοχικά αντίθετα αποτελέσματα για να αλλάξει η πρόβλεψη . Αυτό σημαίνει ότι ο predictor "αντιστέκεται" στην αλλαγή της πρόβλεψής του βάσει ενός μεμονωμένου αντίθετου αποτελέσματος, φιλτράροντας τον θόρυβο και ευνοώντας σταθερές συμπεριφορές.
- **FSM 2 (Διάγραμμα 2):** Είναι πιο επιρρεπή προς την κατάσταση SN και λιγότερο προς την ST. Ευνοεί διακλαδώσεις που γίνονται σταθερά Not Taken.

- **FSM 3 (Διάγραμμα 3):** Το FSM 3 αντίστοιχα είναι επιρρεπή ως προς την κατάσταση ST και λιγότερο επιθετικό προς την SN , δηλαδή χρειάζονται δύο NT για να πάει από WT σε SN). Αυτό ευνοεί τις διακλαδώσεις που γίνονται σταθερά Taken.

- **FSM 4 (Διάγραμμα 4):** Αυτό το FSM είναι πιο "επιθετικό" ως προς την επίτευξη και των δύο ισχυρών καταστάσεων (συνδυασμός FSM 2 και FSM 3). Αυτή η στρατηγική επιταχύνει την προσαρμογή σε μια νέα, σταθερή συμπεριφορά (είτε Taken είτε Not Taken) αλλά μπορεί να είναι επιζήμια αν η συμπεριφορά της διακλάδωσης αλλάζει συχνά.

- **FSM 5 (Διάγραμμα 5):** Αυτό το FSM παρουσιάζει μια μη τυπική συμπεριφορά: ενώ επιτρέπει τη μετάβαση στην Strongly Taken κατάσταση (ST[11]), ένα επόμενο 'Taken' αποτέλεσμα οδηγεί τον μετρητή πίσω στην Weakly Taken κατάσταση (WT[10]). Αυτό εμποδίζει τον predictor να "κλειδώσει" στην ST κατάσταση, ακόμα και για διακλαδώσεις που είναι συνεχώς Taken. Αυτή η αδυναμία διατήρησης της ισχυρής κατάστασης Taken πιθανότατα εξηγεί γιατί αυτό το FSM έχει, σύμφωνα με τα αποτελέσματα, τη χειρότερη απόδοση.

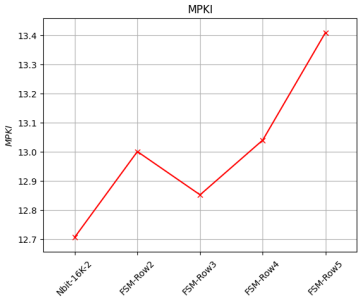
Με βάση τα αποτελέσματα των benchmarks, όπου ο standard 2-bit κορεσμένος μετρητής (Row 1) αποδίδει γενικά καλύτερα με τα εναλλακτικά FSMs, και λαμβάνοντας υπόψη την απλότητά του, **θα επιλέγαμε τον standard 2-bit saturating counter (Row 1)** ως τον προτιμώμενο predictor μεταξύ των εξεταζόμενων 2-bit σχημάτων. Τα εναλλακτικά FSMs, αν και θεωρητικά προσφέρουν διαφορετικές στρατηγικές, δεν φαίνεται να παρέχουν σταθερό πλεονέκτημα απόδοσης που να δικαιολογεί την υιοθέτησή τους έναντι του καθιερωμένου και καλά μελετημένου standard 2-bit counter.

Figure 2: Table VI: Εναλλακτικά FSMs

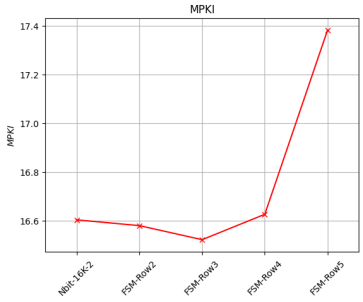
TABLE VI
BEST FIVE MACHINES IGNORING STARTING STATE

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

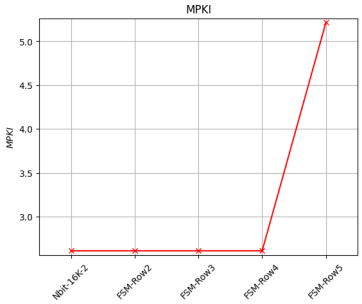
Figure 3: Απόδοση (MPKI) N-bit predictors (N=2) για διαφορετικα FSM



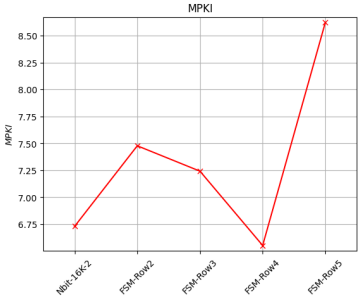
(a) 401.bzip2



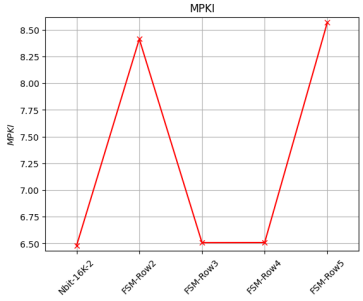
(b) 403.gcc



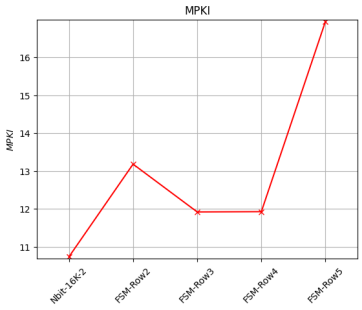
(c) 410.bwaves



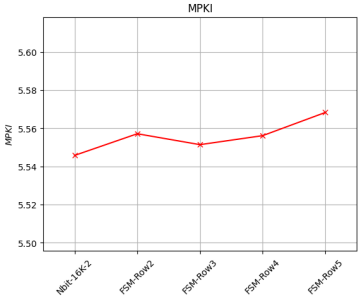
(d) 416.gamess



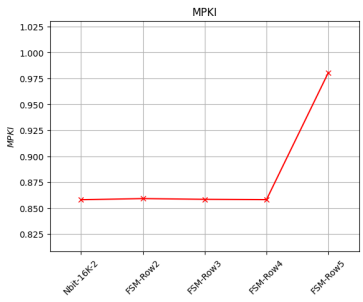
(e) 429.mcf



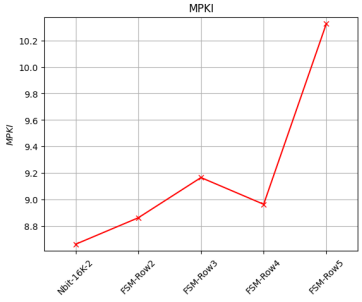
(f) 433.milc



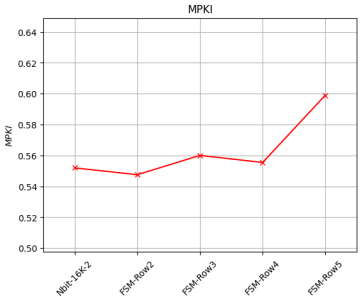
(g) 436.cactusADM



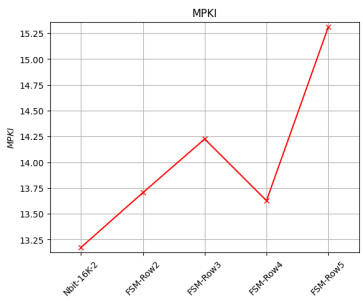
(h) 437.leslie3d



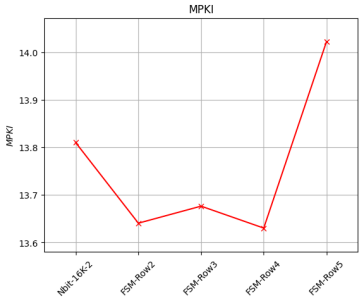
(i) 450.soplex



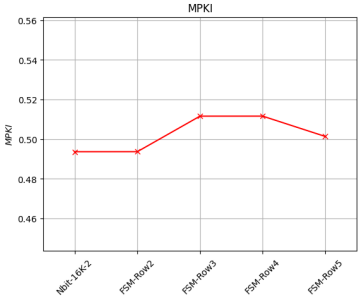
(j) 456.hmmer



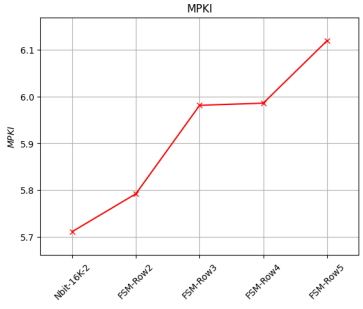
(k) 459.GemsFDTD



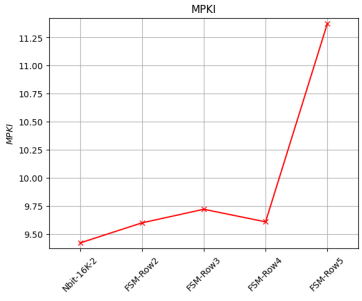
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

3 - Σταθερό Hardware Budget 32K bits (Ερώτημα iii)

Στο τρίτο αυτό ερώτημα, η σύγκριση των predictors γίνεται υπό τον περιορισμό ενός σταθερού συνολικού hardware budget για τον πίνακα BHT, ο οποίος ορίζεται σε 32K bits = 32768 bits. Το συνολικό μέγεθος του BHT δίνεται από τον τύπο:

$$\text{Μέγεθος (bits)} = \text{Αριθμός Καταχωρήσεων} \times N$$

όπου N είναι ο αριθμός των bits ανά καταχώρηση.

Για να διατηρηθεί το μέγεθος σταθερό στα 32K bits, ο αριθμός των καταχωρήσεων (και συνεπώς τα *index_bits*) πρέπει να προσαρμοστεί ανάλογα με το N:

- Για $N = 1$: Αριθμός Καταχωρήσεων $= 32768/1 = 32768 = 2^{15}$. Άρα, $index\_bits = 15$.
- Για $N = 2$: Αριθμός Καταχωρήσεων $= 32768/2 = 16384 = 2^{14}$. Άρα, $index\_bits = 14$.
- Για $N = 4$: Αριθμός Καταχωρήσεων $= 32768/4 = 8192 = 2^{13}$. Άρα, $index\_bits = 13$.

Εξετάζουμε τους N-bit predictors για $N = 1, 2, 4$ με τις παραπάνω διαμορφώσεις, καθώς και τα εναλλακτικά FSMs (Rows 2-5) για την περίπτωση $N = 2$.

Η αρχικοποίηση αυτών των 7 συνολικά predictors (N=1, N=2 Rows 1-5, N=4) στο `cslab_branch.cpp` φαίνεται στο παρακάτω κώδικα 5.

Listing 5: Αρχικοποίηση N-bit predictors (N=1, 2, 4 με 32K bits hardware)

```
VOID InitPredictors(){
/* Question 5.3 (iii) */
// - N-bit predictors with constant hardware 32K bits -
// N=1bit -> index_bits=15
branch_predictors.push_back(new NbitPredictor(15, 1));
// N=2bit -> index_bits=14
branch_predictors.push_back(new NbitPredictor(14, 2));
// N=4bit -> index_bits=13
branch_predictors.push_back(new NbitPredictor(13, 4));

// - For N=2 also the alternative FSMs (rows 2-5) -
for (unsigned r = 2; r <= 5; ++r) {
    branch_predictors.push_back(new FSMPredictor(r));
}
}
```

3.1 - Συμπεράσματα και Επιλογή Predictor (N=1, 2, 4, 32K bits hardware)

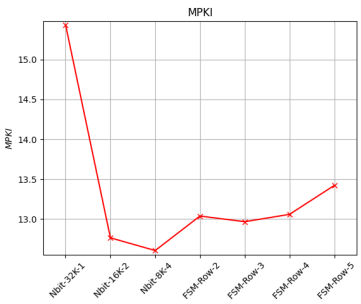
Η διατήρηση σταθερού hardware στα 32K bits εισάγει ένα σημαντικό trade-off: η αύξηση των bits ανά καταχώρηση (N) για βελτίωση της τοπικής ακρίβειας πρόβλεψης οδηγεί σε μείωση του συνολικού αριθμού των καταχωρήσεων στον BHT, αυξάνοντας την πιθανότητα συγκρούσεων, όπου διαφορετικές διακλαδώσεις αντιστοιχίζονται στην ίδια καταχώρηση. Η σύγκριση της απόδοσης (MPKI) των 7 εξεταζόμενων predictors (N=1 με 32K entries, N=2 Rows 1-5 με 16K entries, N=4 με 8K entries), όπως φαίνεται στο Σχήμα [Σχήμα, 4], οδηγεί στις παρακάτω παρατηρήσεις:

- **Απόδοση N=1 (32K entries):** Όπως ήταν αναμενόμενο και από το ερώτημα 5.3(i), ο 1-bit predictor, παρότι διαθέτει τον μεγαλύτερο αριθμό καταχωρήσεων (32K), παρουσιάζει γενικά την υψηλότερη τιμή MPKI (χειρότερη απόδοση) λόγω της περιορισμένης ικανότητάς του να αποθηκεύει ιστορικότητα.
- **Απόδοση N=2 (16K entries, Rows 1-5):** Οι 2-bit predictors με 16K entries αποδίδουν σημαντικά καλύτερα από τον N=1. Μεταξύ των 2-bit σχημάτων, η σχετική κατάταξη είναι παρόμοια με αυτή που παρατηρήθηκε στο ερώτημα 5.3(ii), με τον standard μετρητή (Row 1) και τα FSMs 3 και 4 να είναι συνήθως οι ισχυρότεροι ανταγωνιστές σε αυτή την κατηγορία.

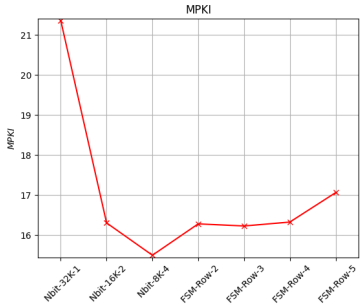
- **Απόδοση N=4 (8K entries):** Ο 4-bit predictor, παρότι διαθέτει μόνο 8192 (8K) καταχωρήσεις (τις λιγότερες από όλους τους προηγούμενους), φαίνεται να επιτυγχάνει το χαμηλότερο MPKI (καλύτερη απόδοση) στα περισσότερα benchmarks. Αυτό υποδηλώνει ότι, για το συγκεκριμένο budget των 32K bits και για αυτά τα benchmarks, η αυξημένη ακρίβεια που προσφέρουν τα 4 bits ανά καταχώρηση υπεραντισταθμίζει το μειονέκτημα του μικρότερου αριθμού καταχωρήσεων και των πιθανώς αυξημένων συγκρούσεων.

Λαμβάνοντας υπόψη τον σταθερό hardware των 32K bits και τα αποτελέσματα των προσομοιώσεων, ο **4-bit predictor (N=4) με 8K καταχωρήσεις** αναδεικνύεται ως η βέλτιστη επιλογή. Αυτός ο predictor συνδυάζει την ικανότητα αποθήκευσης περισσότερης ιστορικότητας (4 bits ανά καταχώρηση) με έναν επαρκή αριθμό καταχωρήσεων (8K), προσφέροντας έτσι την καλύτερη συνολική απόδοση (MPKI) στα περισσότερα benchmarks.

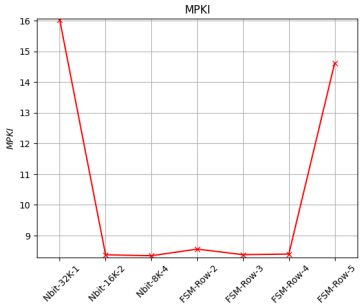
Figure 4: Απόδοση (MPKI) N-bit predictors (N=1, 2, 4) με σταθερό hardware 32K bits



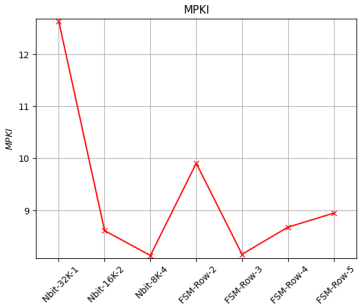
(a) 401.bzip2



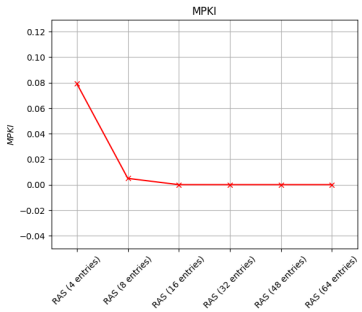
(b) 403.gcc



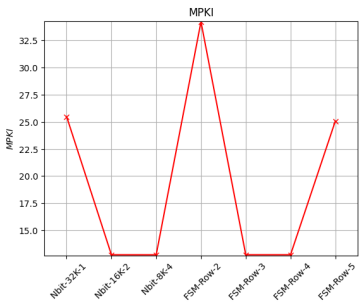
(c) 410.bwaves



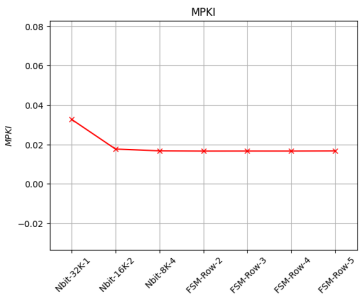
(d) 416.gamess



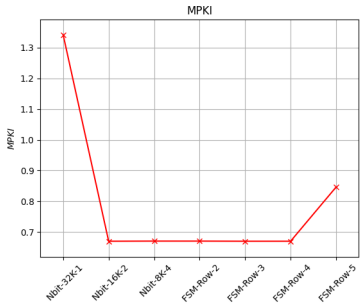
(e) 429.mcf



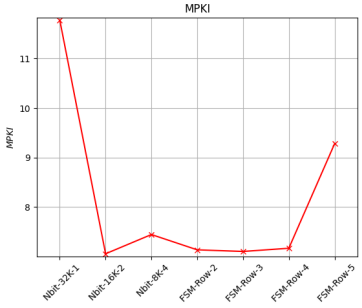
(f) 433.milc



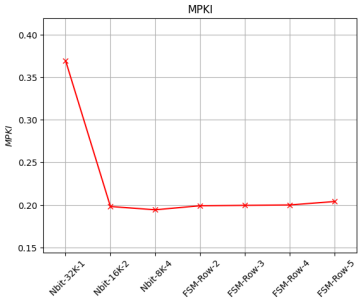
(g) 436.cactusADM



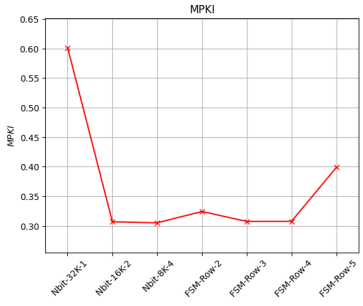
(h) 437.leslie3d



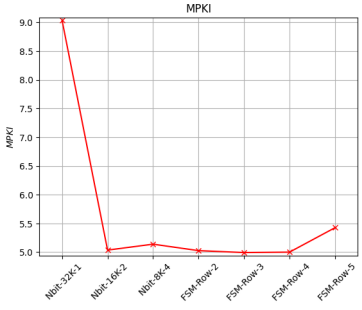
(i) 450.soplex



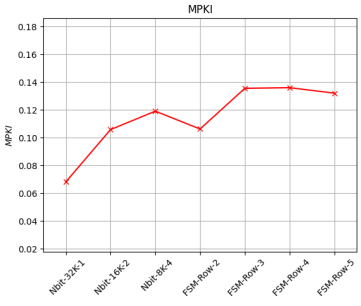
(j) 456.hmmer



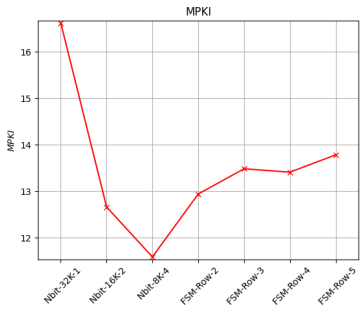
(k) 459.GemsFDTD



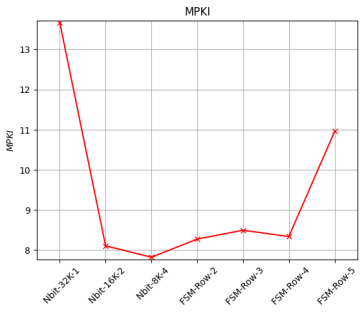
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

Υλοποίηση BTB (Άσκηση 5.4)

Όπως σημειώνεται στην εκφώνηση, υπάρχουν δύο κύριες περιπτώσεις λανθασμένης πρόβλεψης που σχετίζονται με τον BTB:

- **Direction Misprediction:** Η κατεύθυνση της διακλάδωσης (Taken/Not Taken) προβλέφθηκε λανθασμένα.
- **Target Misprediction:** Η κατεύθυνση προβλέφθηκε σωστά (hit), αλλά η διεύθυνση στόχου που δόθηκε από τον BTB ήταν λανθασμένη (κυρίως για έμμεσες διακλαδώσεις - indirect branches).

1 - Θεωρητικές Προσδοκίες

Η απόδοση ενός BTB επηρεάζεται κυρίως από δύο παραμέτρους:

- **Επίδραση Αριθμού Καταχωρήσεων (Entries):**
  - *Αύξηση Entries:* Γενικά, η αύξηση του αριθμού των καταχωρήσεων βελτιώνει την απόδοση του BTB, αυξάνοντας το ποσοστό επιτυχίας (hit rate). Ένας μεγαλύτερος BTB μπορεί να αποθηκεύσει πληροφορία για περισσότερες διακλαδώσεις, μειώνοντας τα capacity misses και, ανάλογα με τη συσχέτιση, τα conflict misses.
  - *Φθίνουσα Απόδοση:* Η βελτίωση από την προσθήκη καταχωρήσεων συνήθως παρουσιάζει φθίνουσα απόδοση.
- **Επίδραση Βαθμού Συσχέτισης (Associativity):**
  - *Αύξηση Associativity (για σταθερά entries):* Η αύξηση της συσχέτισης (π.χ., από 1-way direct mapped σε 2-way ή 4-way set-associative) μειώνει τα conflict misses. Αυτό συμβαίνει όταν πολλαπλές ενεργές διακλαδώσεις αντιστοιχίζονται στο ίδιο set του BTB. Υψηλότερη συσχέτιση επιτρέπει σε περισσότερες από αυτές τις διακλαδώσεις να συνυπάρχουν στο BTB.
  - *Trade-offs:* Η υψηλότερη συσχέτιση αυξάνει την πολυπλοκότητα της λογικής σύγκρισης, μπορεί να αυξήσει ελαφρώς τον χρόνο πρόσβασης και την κατανάλωση ενέργειας.
  - *Φθίνουσα Απόδοση:* Όπως και με τις καταχωρήσεις, η ωφέλεια από την αύξηση της συσχέτισης είναι φθίνουσα. Η μετάβαση από direct-mapped (1-way) σε 2-way συχνά δίνει σημαντικό όφελος, αλλά η μετάβαση από 4-way σε 8-way μπορεί να δώσει μικρότερη βελτίωση, ειδικά αν τα conflict misses δεν είναι ο κυρίαρχος παράγοντας αστοχιών.

2 - Υλοποίηση του BTB

Παρακάτω δίνεται η υλοποίηση του BTB (Branch Target Buffer) στο `cslab_branch.cpp`. Η υλοποίηση περιλαμβάνει την προσθήκη ενός πίνακα BTB, ο οποίος αποθηκεύει τις διευθύνσεις στόχου των διακλαδώσεων και τις αντίστοιχες καταχωρήσεις τους.

Listing 6: Υλοποίηση BTB στο cslab_branch.cpp

```
class BTBPredictor : public BranchPredictor
{
public:
    BTBPredictor(int btb_lines, int btb_assoc)
        : BranchPredictor(), table_lines(btb_lines), table_assoc(btb_assoc), numSets(table_lines / table_assoc), sets(numSets,
              std::vector<BTBEntry>(table_assoc)),
          current_time(0), NumCorrectTargetPredictions(0) {}

    ~BTBPredictor() {}

    virtual bool predict(ADDRINT ip, ADDRINT target){
        int index = ip & (numSets - 1);
        for (auto &entry : sets[index]){
            if (entry.valid && entry.ip == ip)
                return true;
        }
        return false;
    }

    virtual void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target){
        int index = ip & (numSets - 1);
        bool invalid_found = false;
        BTBEntry *lru_entry = nullptr;

        if (predicted){
            for (auto &entry : sets[index]){
                if (entry.ip == ip){
                    if (actual){
                        entry.timestamp = current_time++;
                        if (entry.target != target)
                            entry.target = target;
                        else
                            NumCorrectTargetPredictions++;
                        break;
                    }

                    else{
                        entry.valid = false;
                        break;
                    }
                }
            }
        }
        else {
            if (actual){
                // Insert new entry
                for (auto &entry : sets[index]){
                    if (!entry.valid){
                        entry.valid = true;
                        entry.ip = ip;
                        entry.target = target;
                        entry.timestamp = current_time++;
                        invalid_found = true;
                        break;
                    }

                    if (!lru_entry || entry.timestamp < lru_entry->timestamp)
                        lru_entry = &entry;
                }

                if (!invalid_found){
                    lru_entry->valid = true;
                    lru_entry->ip = ip;
                    lru_entry->target = target;
                    lru_entry->timestamp = current_time++;
                }
            }
        }

        updateCounters(predicted, actual);
    }

    virtual string getName(){
        std::ostringstream stream;
        stream << "BTB-" << table_lines << "-" << table_assoc;
        return stream.str();
    }

    UINT64 getNumCorrectTargetPredictions(){
        return NumCorrectTargetPredictions;
    }

private:
    int table_lines, table_assoc, numSets;
    struct BTBEntry{
        bool valid = false;
        ADDRINT ip = 0;
        ADDRINT target = 0;
        uint64_t timestamp = 0;
    };
    std::vector<std::vector<BTBEntry>> sets;
    uint64_t current_time;
    UINT64 NumCorrectTargetPredictions;
};
```

3 - Εκτέλεση Benchmark για BTB Predictor

Προσομοιώθηκαν οι ακόλουθοι συνδυασμοί μεγέθους και συσχέτισης του BTB:

- 512 entries: 1-way, 2-way
- 256 entries: 2-way, 4-way
- 128 entries: 2-way, 4-way
- 64 entries: 4-way, 8-way

Στο cslab_branch.cpp, μέσω της συνάρτησης `InitPredictors()`, αρχικοποιούνται οι παραπάνω διαμορφώσεις του BTB ως εξής:

Listing 7: Αρχικοποίηση BTB predictors

```
VOID InitPredictors(){
    btb_predictors.push_back(new BTBPredictor(512, 1)); // 512 lines, 1-way
    btb_predictors.push_back(new BTBPredictor(512, 2)); // 512 lines, 2-way
    btb_predictors.push_back(new BTBPredictor(256, 2)); // 256 lines, 2-way
    btb_predictors.push_back(new BTBPredictor(256, 4)); // 256 lines, 4-way
    btb_predictors.push_back(new BTBPredictor(128, 2)); // 128 lines, 2-way
    btb_predictors.push_back(new BTBPredictor(128, 4)); // 128 lines, 4-way
    btb_predictors.push_back(new BTBPredictor(64, 4)); // 64 lines, 4-way
    btb_predictors.push_back(new BTBPredictor(64, 8)); // 64 lines, 8-way
}
```

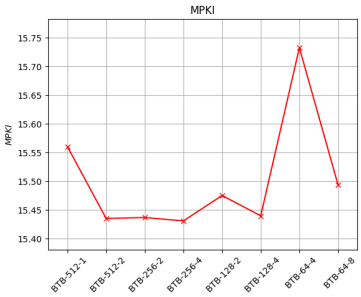
Για κάθε διαμόρφωση, εκτελέστηκαν τα benchmarks με τα ref inputs και καταγράφηκε η μετρική MPKI, η οποία αντικατοπτρίζει την επίδραση της ακρίβειας του BTB (και του σχετιζόμενου direction predictor) στην συνολική απόδοση. Από την ανάλυση των αποτελεσμάτων (Σχήμα 5 και τα συνολικά δεδομένα):

- **Βέλτιστη Διαμόρφωση:** Επιβεβαιώνεται η παρατήρηση ότι η διαμόρφωση με **512 entries και 2-way associativity** επιτυγχάνει γενικά τη χαμηλότερη τιμή MPKI, καθιστώντας την την πιο αποδοτική από τις εξεταζόμενες.
- **Δεύτερη Καλύτερη Διαμόρφωση:** Η διαμόρφωση με **256 entries και 4-way associativity** ακολουθεί συχνά ως η δεύτερη καλύτερη επιλογή.
- **Γενικές Τάσεις:** Όπως αναμενόταν, η αύξηση των entries (π.χ., σύγκριση 128/2-way vs 256/2-way vs 512/2-way) και η αύξηση της associativity (π.χ., σύγκριση 256/2-way vs 256/4-way) τείνουν να βελτιώνουν την απόδοση (μειώνουν το MPKI), αν και ο βαθμός βελτίωσης διαφέρει ανά benchmark.

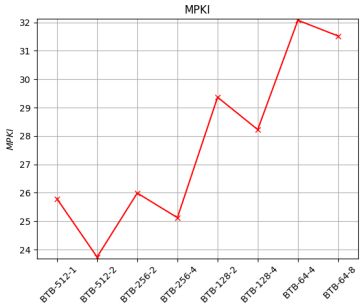
4 - Επιλογή Βέλτιστων Παραμέτρων για BTB

Με βάση τα αποτελέσματα των προσομοιώσεων και την ανάλυση του MPKI για τα διάφορα benchmarks, η βέλτιστη οργάνωση για τον BTB από αυτές που εξετάστηκαν είναι αυτή με **512 καταχωρήσεις και 2-way set associativity**. Αυτή η διαμόρφωση παρέχει τον καλύτερο συνδυασμό χωρητικότητας και διαχείρισης συγκρούσεων, οδηγώντας στη χαμηλότερη μέση τιμή MPKI και είναι σύμφωνη με την θεωρία που αναλύσαμε.

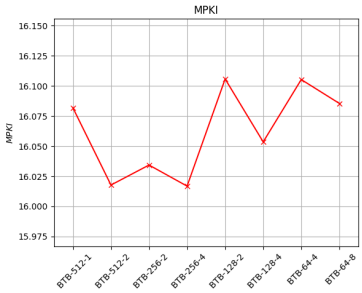
Figure 5: Απόδοση (MPKI) BTB



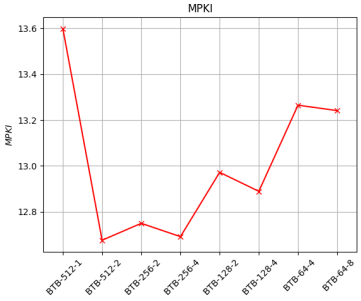
(a) 401.bzip2



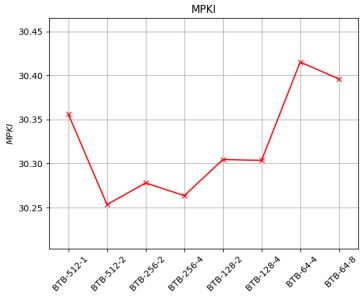
(b) 403.gcc



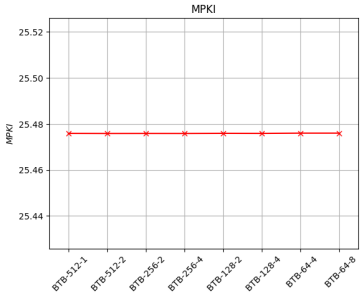
(c) 410.bwaves



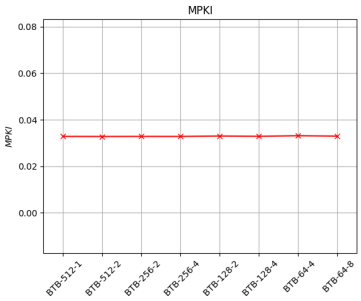
(d) 416.gamess



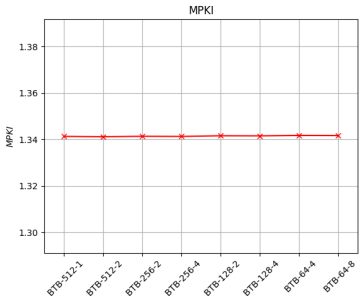
(e) 429.mcf



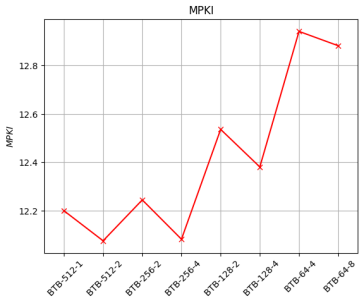
(f) 433.milc



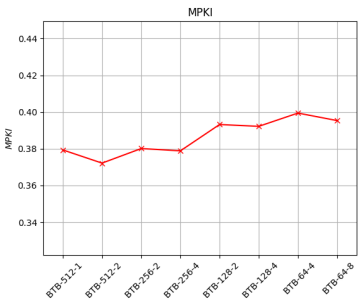
(g) 436.cactusADM



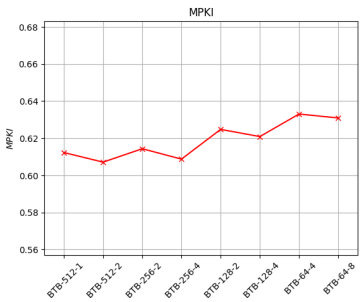
(h) 437.leslie3d



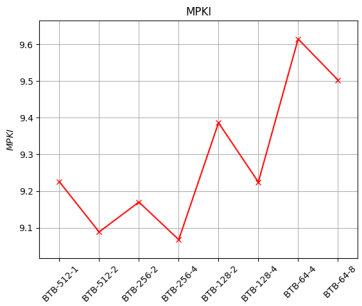
(i) 450.soplex



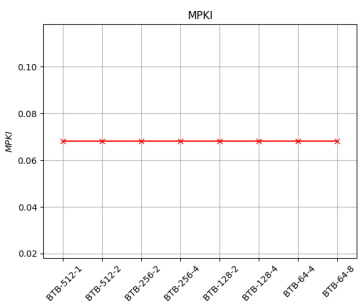
(j) 456.hmmer



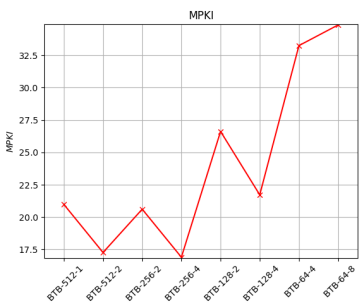
(k) 459.GemsFDTD



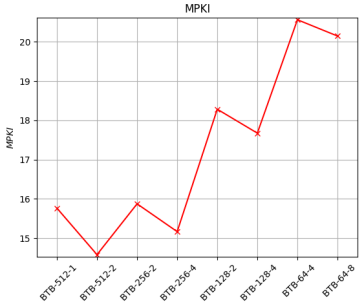
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

Μελέτη RAS (Άσκηση 5.5)

1 - Λειτουργία RAS και Επίδραση Μεγέθους

Η Return Address Stack (RAS) είναι μια μικρή, εξειδικευμένη δομή υλικού (hardware stack) που χρησιμοποιείται για την ακριβή και γρήγορη πρόβλεψη της διεύθυνσης στόχου των εντολών επιστροφής από υπορουτίνες/συναρτήσεις (”return” instructions) αφιερωμένη στην πρόβλεψη της διεύθυνσης στόχου των εντολών επιστροφής από συναρτήσεις (‘return‘). Η αρχή λειτουργίας της είναι απλή και βασίζεται στη δυναμική συμπεριφορά των κλήσεων συναρτήσεων:

- Όταν εκτελείται μια εντολή ‘call‘, η διεύθυνση της αμέσως επόμενης εντολής ωθείται (push) στην κορυφή της RAS.
- Όταν συναντάται μια εντολή ‘return‘, το υλικό προβλέπει ότι η διεύθυνση στόχου θα είναι αυτή που βρίσκεται στην κορυφή της RAS. Η διεύθυνση αυτή εξάγεται (pop) από τη RAS και ο επεξεργαστής ξεκινά την ανάκτηση εντολών (fetch) από αυτή τη διεύθυνση.

Η RAS είναι ευάλωτη σε υπερχείλιση (overflow), καθώς αν το βάθος κλήσεων υπερβεί το βάθος της RAS, παλαιότερες διευθύνσεις επιστροφής χάνονται. Αυτό οδηγεί σε misprediction όταν εκτελεστεί η αντίστοιχη εντολή `return`.

Στην άσκηση αυτή, θα μελετηθεί η επίδραση της μεταβολής του αριθμού των εγγραφών της RAS, εξετάζοντας τα μεγέθη 4, 8, 16, 32, 48 και 64 εγγραφές.

```
VOID InitRas(){
   ras_vec.push_back(new RAS(4));
   ras_vec.push_back(new RAS(8));
   ras_vec.push_back(new RAS(16));
   ras_vec.push_back(new RAS(32));
   ras_vec.push_back(new RAS(48));
   ras_vec.push_back(new RAS(64));
}
```

Από την ανάλυση των αποτελεσμάτων [Σχήμα, 6], παρατηρούνται τα εξής:

- **Γενική Τάση:** Η αύξηση του αριθμού εγγραφών στη RAS μειώνει τις αστοχίες, καθώς περισσότερες κλήσεις συναρτήσεων αποθηκεύονται σωστά.
- **Φθίνουσα Απόδοση:** Μετά από ένα σημείο, η βελτίωση από την αύξηση του μεγέθους της RAS γίνεται αμελητέα.

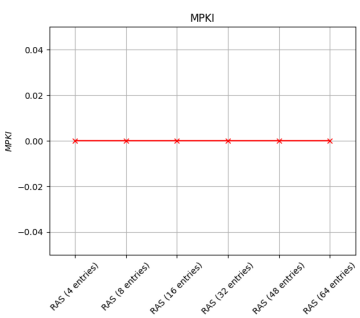
2 - Επιλογή Κατάλληλου Μεγέθους RAS

Θέλουμε ένα μέγεθος αρκετά μεγάλο ώστε να καλύπτει τις ανάγκες των περισσότερων benchmarks και να επιτυγχάνει χαμηλό ποσοστό αστοχίας, αλλά όχι υπερβολικά μεγάλο ώστε να σπαταλά πόρους υλικού χωρίς ανάλογο όφελος στην απόδοση.

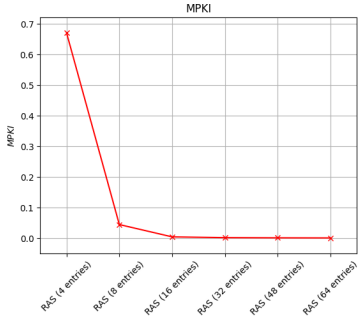
Με βάση τα αποτελέσματα παρατηρούμε ότι καθώς αυξάνεται ο αριθμός των εγγραφών της RAS, η απόδοση (MPKI) βελτιώνεται. Ωστόσο, η βελτίωση είναι φθίνουσα και η αύξηση του αριθμού των εγγραφών πέρα ένα σημείο προσφέρει περιορισμένα οφέλη. Για αυτόν τον λόγο, **επιλέγουμε 16 εγγραφές** για τη RAS, καθώς προσφέρουν καλή ισορροπία μεταξύ απόδοσης και κόστους υλικού. *Δεν επιλέγουμε* 8 εγγραφές, καθώς παρατηρούμε ότι σε ορισμένα προγράμματα υπάρχει σημαντική μείωση του MPKI από τις 8 στις 16 εγγραφές[1].

[1]Παρατηρούμε τα benchmarks `471.omnetpp` και `483.xalancbmk` να έχουν σχετικά μεγάλη επίδραση στην απόδοση απο 8 entries σε 16. Αυτό συμβαίνει καθώς όπως τονίσαμε και στο ερώτημα 5.2 στα στατιστικά δεδομένα, βλέπουμε μεγάλο ποσοστό στα **Returns** και **Calls**

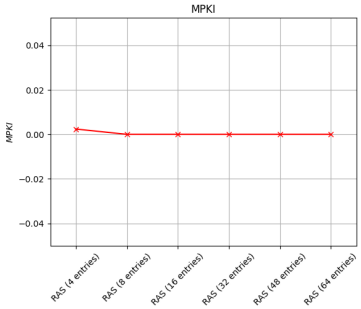
Figure 6: Απόδοση (MPKI) RAS



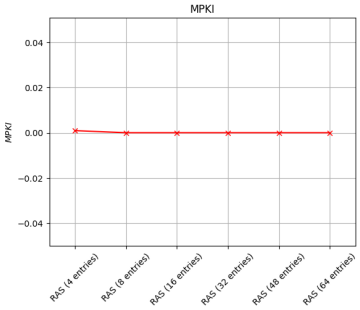
(a) 401.bzip2



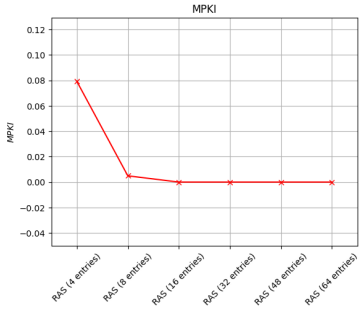
(b) 403.gcc



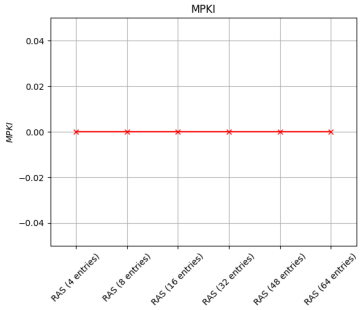
(c) 410.bwaves



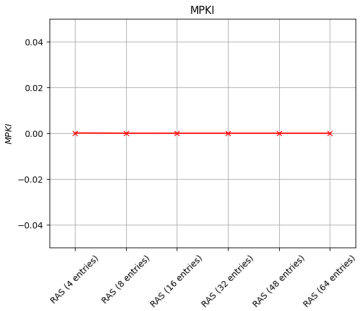
(d) 416.gamess



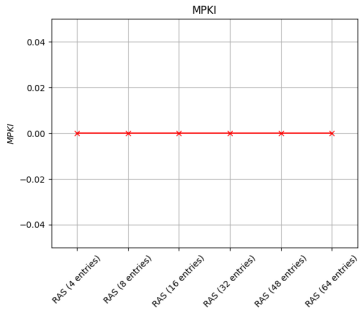
(e) 429.mcf



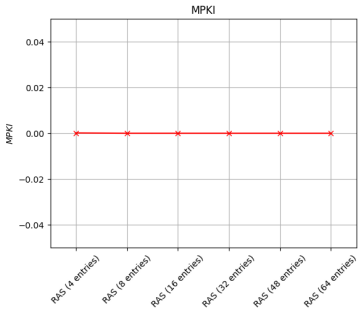
(f) 433.milc



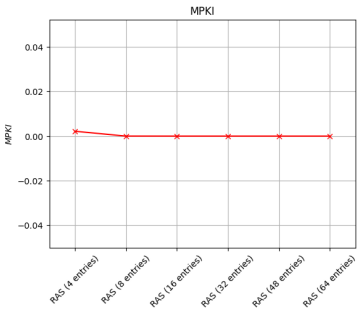
(g) 436.cactusADM



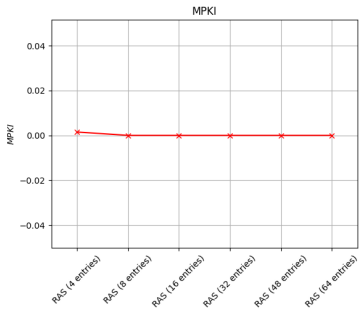
(h) 437.leslie3d



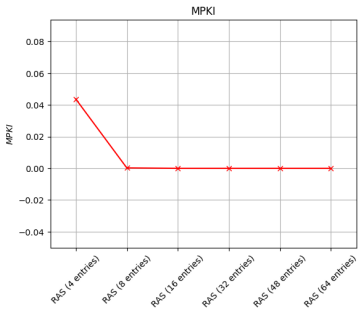
(i) 450.soplex



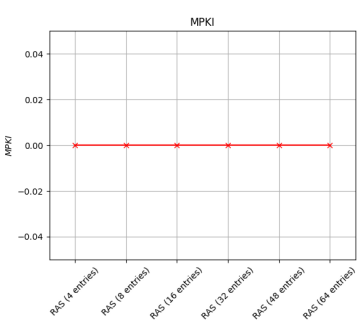
(j) 456.hmmer



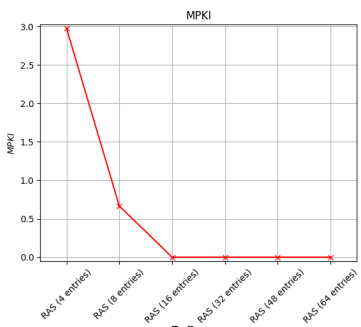
(k) 459.GemsFDTD



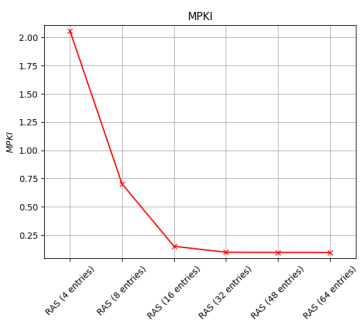
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

Σύγκριση Διαφορετικών Predictors (Άσκηση 5.6)

Σε αυτή την ενότητα, πραγματοποιείται σύγκριση της απόδοσης (μετρούμενης σε MPKI) ενός ευρέος φάσματος branch predictors, από απλούς στατικούς μέχρι σύνθετους δυναμικούς και υβριδικούς, χρησιμοποιώντας τα ref inputs.

1 - Static Always Taken

Περιγραφή: Ο πιο απλός στατικός predictor. Προβλέπει ότι όλες οι διακλαδώσεις υπό συνθήκη (conditional branches) θα εκτελεστούν (Taken). Δεν απαιτεί καμία αποθήκευση ιστορικού ή κατάστασης.
Παράμετροι: Δεν υπάρχουν παράμετροι διαμόρφωσης.
Hardware Overhead: Αμελητέο.

Listing 8: Υλοποίηση Static Always Taken Predictor

```
class StaticAlwaysTakenPredictor : public BranchPredictor
{
    public:
    // Constructor
    StaticAlwaysTakenPredictor() : BranchPredictor() {};

    // Destructor
    ~StaticAlwaysTakenPredictor() {};

    // predict(): Always predicts Taken (true)
    virtual bool predict(ADDRINT ip, ADDRINT target)
    {
           return true;
       }

    // update(): Updates counters but doesn't change predictor state
    virtual void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target)
    {
           // Update the prediction counters in the base class
           updateCounters(predicted, actual);

           // Static predictor, state does not change.
           // ip and target arguments are ignored.
       }

    // getName(): Returns the name of the predictor
    virtual string getName()
    {
           return "Static-AlwaysTaken";
       }
};
```

2 - Static BTFNT

Περιγραφή: Static Backward Taken, Forward Not Taken. Ένας στατικός predictor που εκμεταλλεύεται την παρατήρηση ότι οι διακλαδώσεις προς τα πίσω (backward branches) είναι συχνά Taken, ενώ οι διακλαδώσεις προς τα εμπρός (forward branches) είναι συχνά Not Taken. Η πρόβλεψη βασίζεται στη διεύθυνση στόχου σε σχέση με τη διεύθυνση της διακλάδωσης.
Παράμετροι: Δεν υπάρχουν παράμετροι διαμόρφωσης.
Hardware Overhead: Αμελητέο.

Listing 9: Υλοποίηση Static BTFNT Predictor

```
class StaticBTFNTPredictor : public BranchPredictor
{
public:
    StaticBTFNTPredictor() : BranchPredictor() {}
    ~StaticBTFNTPredictor() {}

    virtual bool predict(ADDRINT ip, ADDRINT target)
    {
        return target < ip;
    }

```

```
    virtual void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target)
    {
        updateCounters(predicted, actual);
    }

    virtual string getName()
    {
        std::ostringstream stream;
        stream << "BTFNT";
        return stream.str();
    }
};
```

3 - N-bit Predictor (Βέλτιστος από 5.3.iii)

Περιγραφή: Ο N-bit predictor που επιλέχθηκε ως βέλτιστος στο ερώτημα 5.3.iii, λαμβάνοντας υπόψη το σταθερό hardware budget των 32K bits. Βασίζεται σε έναν πίνακα BHT όπου κάθε καταχώρηση είναι ένας N-bit κορεσμένος μετρητής.
Παράμετροι: Σύμφωνα με τα αποτελέσματα του 5.3.iii, ο βέλτιστος ήταν ο:

- N = 4 bits
- Αριθμός Καταχωρήσεων BHT = 8192 (2^{13}, index_bits=13)

Hardware Overhead: $8192 \times 4 = 32768$ bits = 32K bits.

4 - Pentium-M Predictor

Περιγραφή: Ένας υβριδικός predictor εμπνευσμένος από την αρχιτεκτονική του Intel Pentium M. Συνήθως συνδυάζει πληροφορίες τοπικού ιστορικού (local history), καθολικού ιστορικού (global history) και πιθανώς πληροφορία πορείας (path information) για να κάνει την τελική πρόβλεψη. Η ακριβής υλοποίηση δίνεται στον κώδικα.
Παράμετροι: Οι εσωτερικές παράμετροι (μεγέθη πινάκων, μήκη ιστορικού) είναι συνήθως καθορισμένες στην υλοποίηση ώστε να προσεγγίζουν το δεδομένο overhead.
Hardware Overhead: Περίπου 30K bits (όπως δίνεται στην εκφώνηση).

5 - Local-History Two-Level Predictors

Περιγραφή: Προγνώστες δύο επιπέδων που χρησιμοποιούν τοπικό ιστορικό. Έχουμε πως κάθε καταχώρηση στον πρώτο πίνακα (Branch History Table - BHT) αποθηκεύει το πρόσφατο ιστορικό της συγκεκριμένης διακλάδωσης. Αυτό το τοπικό ιστορικό χρησιμοποιείται για να δεικτοδοτήσει τον δεύτερο πίνακα (Pattern History Table - PHT), ο οποίος περιέχει 2-bit κορεσμένους μετρητές που προβλέπουν την επόμενη έκβαση βάσει του τοπικού μοτίβου. Ο BHT δεικτοδοτείται από τη διεύθυνση της διακλάδωσης (PC).
Παράμετροι:

- PHT entries = 8192 (σταθερό)
- PHT counter length = 2 bits (σταθερό) $\implies$ PHT size = $8192 \times 2 = 16384$ bits (16K bits).
- Συνολικό Hardware Budget = 32K bits (σταθερό).
- Απομένουν $32768 - 16384 = 16384$ bits (16K bits) για τον BHT.
- BHT size = BHT entries (X) $\times$ BHT entry length (Z) = 16384 bits.

Υπολογίζουμε το Z για τις τρεις περιπτώσεις του X:

1. **Local-2K (X=2048):** $Z = 16384/2048 = 8$ bits/entry. *Config: BHT(2K entries, 8 bits/entry), PHT(8K entries, 2 bits/entry)*
2. **Local-4K (X=4096):** $Z = 16384/4096 = 4$ bits/entry. *Config: BHT(4K entries, 4 bits/entry), PHT(8K entries, 2 bits/entry)*
3. **Local-8K (X=8192):** $Z = 16384/8192 = 2$ bits/entry. *Config: BHT(8K entries, 2 bits/entry), PHT(8K entries, 2 bits/entry)*

Hardware Overhead: 32K bits για κάθε μία από τις 3 διαμορφώσεις.

Listing 10: Υλοποίηση Local History Predictor (PAg)

```
class LocalHistoryPredictor : public BranchPredictor
{
private:
    // Predictor Parameters
    const unsigned int bht_entries;  // Number of BHT entries (X)
    const unsigned int history_length; // History length per BHT entry (Z)
    const unsigned int pht_entries;  // Number of PHT entries (fixed at 8192)
    const unsigned int pht_counter_bits; // Bits per PHT counter (fixed at 2)
    const unsigned int pht_index_mask; // Mask for the PHT index (8192 - 1)

    // Mask for hisotry_length (we're keeping only Z bits)
    const uint8_t history_mask;
    // Tables
    std::vector<uint8_t> BHT; // Branch History Table (holds Z bits of history)
    std::vector<uint8_t> PHT; // Pattern History Table (holds 2-bit counters)

    const uint8_t counter_max = 3; // (1 << pht_counter_bits) - 1;
    const unsigned int bht_length; // Lengh BHT (log2(X))

public:
    // Constructor
    LocalHistoryPredictor(unsigned int bht_entries_X, unsigned int history_length_Z, unsigned int pht_entries, unsigned int pht_counter_bits) :
                                                        BranchPredictor(),
                                                        bht_entries(bht_entries_X),
                                                        history_length(history_length_Z),
                                                        pht_entries(pht_entries),
                                                        pht_counter_bits(pht_counter_bits),
                                                        pht_index_mask(pht_entries - 1),
                                                        history_mask((1 << history_length_Z) - 1), // Computation of history_mask
                                                        bht_length(static_cast<unsigned int>(std::round(std::log2(bht_entries_X))))
    {
        // Initialize BHT with 0s (size: X)
        BHT.assign(bht_entries, 0);

        // Initialize PHT with 0 or 1 (size 8192)
        // Initial State Weakly Not Taken (1)
        PHT.assign(pht_entries, 1);
    }

    ~LocalHistoryPredictor() {};

    bool predict(ADDRINT ip, ADDRINT target) override
    {
        unsigned int bht_index = ip % bht_entries;
        uint8_t local_history = BHT[bht_index];

        // Shift the PC component to make room for the local history bits
        unsigned int shifted_pc_component = ip << bht_length;
        unsigned int pht_index = (shifted_pc_component | local_history) & pht_index_mask;
        uint8_t counter_state = PHT[pht_index];
        bool prediction = (counter_state >= 2);
        return prediction;
    }

    void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target) override
    {
        unsigned int bht_index = ip % bht_entries;
        uint8_t local_history = BHT[bht_index];
        unsigned int shifted_pc_component = ip << bht_length;
        unsigned int pht_index = (shifted_pc_component | local_history) & pht_index_mask;

        uint8_t old_counter_state = PHT[pht_index];
        uint8_t new_counter_state;
        if (actual)
        {
            new_counter_state = (old_counter_state < counter_max) ? old_counter_state + 1 : counter_max;
        }
        else
        {
            new_counter_state = (old_counter_state > 0) ? old_counter_state - 1 : 0;
        }
        PHT[pht_index] = new_counter_state;

        unsigned int bht_update_mask = (1 << (bht_length - 1));
```

```
        uint8_t new_history = ((local_history >> 1) | (actual ? bht_update_mask : 0)) & history_mask;
        BHT[bht_index] = new_history;

        updateCounters(predicted, actual);
    }

    std::string getName() override
    {
        std::ostringstream stream;
        stream << "Local-" << bht_entries << "ent-" << history_length << "hist";
        return stream.str();
    }
};
```

6 - Global-History Two-Level Predictors (GAg/gshare)

Περιγραφή: Αποτελεί ειδική περίπτωση του Local History Predictor, όπου το BHT έχει μόνο ένα entry. Αυτός ο μοναδικός καταχωρητής ιστορικού (Branch History Register - BHR) αποθηκεύει τα αποτελέσματα των τελευταίων k διακλαδώσεων που εκτελέστηκαν συνολικά στο πρόγραμμα. Το περιεχόμενο του BHR (σε συνδυασμό με τη διεύθυνση της τρέχουσας διακλάδωσης - PC) χρησιμοποιείται για να δεικτοδοτήσει τον πίνακα PHT (Pattern History Table). Το κόστος του BHR θεωρείται αμελητέο.
Παράμετροι:

- Συνολικό Hardware Budget (για το PHT) = 32K bits (σταθερό).
- PHT size = PHT entries (Z) × PHT counter length (X) = 32768 bits.

Εξετάζονται οι περιπτώσεις για counter length X=2 και X=4, και για κάθε X, μήκος BHR = 2 και 4 bits:

1. **Global-16K-2bit (X=2):** PHT entries $Z = 32768/2 = 16384$.
   - *BHR=2: PHT(16K entries, 2 bits/entry), BHR(2 bits)*
   - *BHR=4: PHT(16K entries, 2 bits/entry), BHR(4 bits)*
2. **Global-8K-4bit (X=4):** PHT entries $Z = 32768/4 = 8192$.
   - *BHR=2: PHT(8K entries, 4 bits/entry), BHR(2 bits)*
   - *BHR=4: PHT(8K entries, 4 bits/entry), BHR(4 bits)*

Hardware Overhead: 32K bits για κάθε μία από τις 4 διαμορφώσεις (αγνοώντας το κόστος BHR).

Listing 11: Υλοποίηση Global History Predictor (gshare/GAg)

```
class GlobalHistoryPredictor : public BranchPredictor
{
private:
    const unsigned int pht_entries;
    const unsigned int cntr_bits;
    const unsigned int bhr_length;

    // Table PHT
    std::vector<uint8_t> PHT; // Pattern History Table (holds X-bit counters)

    // Branch History Register (BHR)
    uint8_t BHR; // holds N bits history

    // Masks and constants
    const uint8_t counter_max;
    const uint8_t bhr_mask;
    const unsigned int pht_index_mask;
    const unsigned int pht_index_bits;

public:
    // Constructor
    GlobalHistoryPredictor(unsigned int pht_entries_Z, unsigned int counter_length_X, unsigned int bhr_length_N) :
```

```
                                                BranchPredictor(),
                                                pht_entries(pht_entries_Z),
                                                cntr_bits(counter_length_X),
                                                bhr_length(bhr_length_N),
                                                BHR(0), // Initialize BHR to 0
                                                counter_max((1 << counter_length_X) - 1),
                                                bhr_mask((1 << bhr_length_N) - 1),
                                                pht_index_mask(pht_entries_Z - 1),
                                                pht_index_bits(static_cast<unsigned int>(std::round(std::log2(pht_entries_Z))))
    {
        // Initialize PHT (size Z)
        PHT.assign(pht_entries, 1); // Initial State: Weakly Not Taken (1)
    }

    ~GlobalHistoryPredictor() {};


    bool predict(ADDRINT ip, ADDRINT target) override
    {
        unsigned int shifted_pc_component = ip << bhr_length;
        unsigned int pht_index = (shifted_pc_component | BHR) & pht_index_mask;
        uint8_t counter_state = PHT[pht_index];

        uint8_t prediction = counter_state >> (cntr_bits - 1);
        return (prediction != 0);
    }

    void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target) override
    {
        unsigned int shifted_pc_component = ip << bhr_length;
        unsigned int pht_index = (shifted_pc_component | BHR) & pht_index_mask;
        uint8_t old_counter_state = PHT[pht_index];
        uint8_t new_counter_state;
        if (actual) // Branch Taken
        {
            new_counter_state = (old_counter_state < counter_max) ? old_counter_state + 1 : counter_max;
        }
        else // Branch Not Taken
        {
            new_counter_state = (old_counter_state > 0) ? old_counter_state - 1 : 0;
        }
        PHT[pht_index] = new_counter_state;

        unsigned int bhr_update_mask = (1 << (bhr_length - 1));
        BHR = ((BHR >> 1) | ((actual) ? bhr_update_mask : 0)) & bhr_mask;

        updateCounters(predicted, actual);
    }

    std::string getName() override
    {
        std::ostringstream stream;
        stream << "Global"
               << "-N" << bhr_length                // BHR Length (N)
               << "-X" << cntr_bits                 // Counter Bits (X)
               << "-" << (pht_entries / 1024) << "KPHT"; // PHT Entries (Z in K)
        return stream.str();
    }
};
```

7 - Alpha 21264 Predictor

Περιγραφή: Ένας προηγμένος υβριδικός predictor που χρησιμοποιήθηκε στον επεξεργαστή Alpha 21264. Συνδυάζει τον local history και τον global history predictor. Ένας meta-predictor (choice predictor) επιλέγει δυναμικά ποια από τις δύο προβλέψεις (local ή global) θα χρησιμοποιηθεί για κάθε διακλάδωση. **Παράμετροι:** Οι εσωτερικές παράμετροι είναι καθορισμένες απο τις διαφάνειες στην υλοποίηση ώστε να προσεγγίζουν το δεδομένο overhead.

- **Meta Predictor (M):** 4K entries, 2 bits/entry.
- **Predictor 0 (P_0):** Local History Predictor (LHP) με 1K entries, PHT 1K entries, 3 bits/entry.
- **Predictor 1 (P_1):** Global History Predictor (GHP) με PHT 4K entries, 2 bits/entry.

Hardware Overhead: Περίπου 29K bits (όπως δίνεται στην εκφώνηση).

Listing 12: Υλοποίηση Alpha 21264 Predictor

```
class Alpha21264Predictor : public TournamentHybridPredictor
{
public:
    Alpha21264Predictor() : TournamentHybridPredictor(12, new LocalHistoryPredictor(1024, 3, 1024, 10), new GlobalHistoryPredictor(4096, 2,
        4)) {}
    ~Alpha21264Predictor() {}

    virtual string getName()
    {
        std::ostringstream stream;
        stream << "Alpha21264";
        return stream.str();
    }
};
```

8 - Tournament Hybrid Predictors

Περιγραφή: Αποτελούνται από δύο διαφορετικούς βασικούς προγνώστες (P0 και P1) και έναν meta-predictor (M). Ο meta-predictor παρακολουθεί την πρόσφατη απόδοση των P0 και P1 και επιλέγει ποιου η πρόβλεψη είναι πιθανότερο να είναι σωστή για την τρέχουσα διακλάδωση.
Παράμετροι:

- Meta-predictor (M): Πίνακας με 1024 ή 2048 καταχωρήσεις, όπου κάθε καταχώρηση είναι ένας 2-bit κορεσμένος μετρητής (αποφασίζει μεταξύ P0/P1). Το overhead του M θεωρείται αμελητέο.
- Βασικοί Predictors (P0, P1): Ο καθένας έχει σταθερό overhead 16K bits (16384 bits). Μπορούν να είναι τύπου N-bit, Local-History ή Global-History, αρκεί να ταιριάζουν στο budget των 16K bits.

Υλοποιήθηκαν οι παρακάτω (τουλάχιστον 4) διαμορφώσεις Tournament:

1. **T1 (N2-8K vs Global-8K, Meta=1K):**
    - P0: N-bit (N=2, 8K entries = 16K bits)
    - P1: Global History (PHT: 8K entries, 2 bits/entry = 16K bits)
    - M: 1024 entries, 2 bits/entry
2. **T2 (Global-8K vs Local-8K, Meta=1K):**
    - P0: Local History (PHT: 8K entries, 2 bits/entry = 16K bits)
    - P1: Global History (PHT: 8K entries, 2 bits/entry = 16K bits)
    - M: 1024 entries, 2 bits/entry
3. **T3 (N2 vs Local-8K, Meta=1K):**
    - P0: N-bit (N=2, 8K entries = 16K bits)
    - P1: Local History (PHT: 8K entries, 2 bits/entry = 16K bits)
    - M: 1024 entries, 2 bits/entry
4. **T4 (N2 vs Global-8K, Meta=2K):**
    - P0: N-bit (N=2, 8K entries = 16K bits)
    - P1: Global History (PHT: 8K entries, 2 bits/entry = 16K bits)

- M: 2048 entries, 2 bits/entry

Hardware Overhead: Περίπου $16K + 16K = 32K$ bits (αγνοώντας το M).

Listing 13: Υλοποίηση Tournament Predictor(s)

```
class TournamentHybridPredictor : public BranchPredictor
{
public:
    TournamentHybridPredictor(unsigned int index_bits_, BranchPredictor *p1, BranchPredictor *p2) : BranchPredictor(),
          index_bits(index_bits_), predictor1(p1), predictor2(p2), table_entries(1 << index_bits)
    {
        TABLE = new unsigned long long[table_entries];
        COUNTER_MAX = 3;
        memset(TABLE, 0, table_entries * sizeof(*TABLE));
    }

    ~TournamentHybridPredictor()
    {
        delete[] TABLE;
        delete predictor1;
        delete predictor2;
    }

    virtual bool predict(ADDRINT ip, ADDRINT target)
    {
        unsigned int ip_table_index = ip % table_entries;
        unsigned long long ip_table_value = TABLE[ip_table_index];
        unsigned long long prediction_bit = (ip_table_value >> 1) & 1;

        if (prediction_bit)
            return (predictor2->predict(ip, target));
        return (predictor1->predict(ip, target));
    }

    virtual void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target)
    {
        unsigned int ip_table_index = ip % table_entries;
        bool prediction1 = predictor1->predict(ip, target);
        bool prediction2 = predictor2->predict(ip, target);

        if (prediction1 != prediction2)
        {
            if (prediction1 == actual && (TABLE[ip_table_index] > 0))
                TABLE[ip_table_index]--;
            else if (TABLE[ip_table_index] < COUNTER_MAX)
                TABLE[ip_table_index]++;
        }
        predictor1->update(prediction1, actual, ip, target);
        predictor2->update(prediction2, actual, ip, target);
        updateCounters(predicted, actual);
    }

    virtual string getName()
    {
        std::ostringstream stream;
        stream << "Tournament-" << predictor1->getName() << "-" << predictor2->getName();
        return stream.str();
    }

private:
    unsigned int index_bits;
    BranchPredictor *predictor1;
    BranchPredictor *predictor2;
    unsigned int table_entries;
    unsigned long long *TABLE;
    unsigned int COUNTER_MAX;
};
```

Παρακάτω μπορούμε να δούμε την εισαγωγή των παραπάνω predictors στην συνάρτηση `InitPredictors()` του `cslab_branch.cpp`:

Listing 14: Εισαγωγή Tournament Predictors

```
VOID InitPredictors(){
    // 1. Static Always Taken Predictor
    branch_predictors.push_back(new StaticAlwaysTakenPredictor());

    // 2. Static BTFNT (BackwardTaken-ForwardNotTaken) Predictor
    branch_predictors.push_back(new StaticBTFNTPredictor());

    // 3. N-Bit Predictor (FSM from Row 3)
    branch_predictors.push_back(new NbitPredictor(13,4));

    // 4. Pentium M Predictor
    branch_predictors.push_back(new PentiumMBranchPredictor());

    // 5-7. Local History Predictors (32K budget)
    branch_predictors.push_back(new LocalHistoryPredictor(2048, 8, 8192, 2)); // X=2048, Z=8
    branch_predictors.push_back(new LocalHistoryPredictor(4096, 4, 8192, 2)); // X=4096, Z=4
    branch_predictors.push_back(new LocalHistoryPredictor(8192, 2, 8192, 2)); // X=8192, Z=2

```

```
    // 8-11. Global History Predictors (32K budget)
    // X=2 => Z=16384
    branch_predictors.push_back(new GlobalHistoryPredictor(16384, 2, 2)); // Z=16K, X=2, N=2
    branch_predictors.push_back(new GlobalHistoryPredictor(16384, 2, 4)); // Z=16K, X=2, N=4

    // X=4 => Z=8192
    branch_predictors.push_back(new GlobalHistoryPredictor(8192, 4, 2)); // Z=8K, X=4, N=2
    branch_predictors.push_back(new GlobalHistoryPredictor(8192, 4, 4)); // Z=8K, X=4, N=4

    // 12. Alpha 21264 Predictor
    branch_predictors.push_back(new Alpha21264Predictor());

    // 13-16. Tournament Hybrid Predictors

    branch_predictors.push_back(
        new TournamentHybridPredictor(
            10,
            new NbitPredictor(13, 2),
            new GlobalHistoryPredictor(8192, 2, 2)));

    branch_predictors.push_back(
        new TournamentHybridPredictor(
            10,
            new GlobalHistoryPredictor(8192, 2, 2),
            new LocalHistoryPredictor(8192, 2, 8192, 2)));

    branch_predictors.push_back(
        new TournamentHybridPredictor(
            10,
            new NbitPredictor(13, 2),
            new LocalHistoryPredictor(8192, 2, 8192, 2)));

    branch_predictors.push_back(
        new TournamentHybridPredictor(
            11,
            new NbitPredictor(13, 2),
            new GlobalHistoryPredictor(8192, 2, 2)));
}
```

Τα αποτελέσματα των παραπάνω predictors είναι διαθέσιμα στο Σχήμα 7.

9 - Συμπεράσματα και επιλογή καλύτερου predictor

Η συγκριτική αξιολόγηση των διαφόρων προγνωστών διακλάδωσης, βάσει της μετρικής MPKI που παρουσιάζεται στο Σχήμα 7, οδηγεί στις ακόλουθες παρατηρήσεις σχετικά με την απόδοσή τους:

1. **Στατικοί Predictors:** Όπως ήταν αναμενόμενο, οι στατικοί προγνώστες (*Static-AlwaysTaken* και *Static-BTFNT*) επιδεικνύουν σταθερά την υψηλότερη τιμή MPKI, δηλαδή τη χειρότερη απόδοση. Η αδυναμία τους να προσαρμοστούν στη δυναμική συμπεριφορά των διακλαδώσεων τους καθιστά μη ανταγωνιστικούς σε σύγκριση με τους δυναμικούς μηχανισμούς.

2. **N-bit Predictor (N=4, 8K entries):** Ο βέλτιστος N-bit predictor από το ερώτημα 5.3.iii (N=4, 8K entries, budget 32K bits) παρουσιάζει σημαντικά καλύτερη απόδοση από τους στατικούς predictors.

3. **Local-History Predictors:** Στις τρεις διαμορφώσεις που εξετάστηκαν (Local-2K, Local-4K, Local-8K, όλες με budget 32K bits), αποδίδουν γενικά χειρότερα από τον N=4 predictor (8K entries). Αυτό υποδηλώνει ότι, για το δεδομένο budget, η στρατηγική του N=4 (λιγότερες καταχωρήσεις, περισσότερα bits ανά καταχώρηση) είναι πιο αποτελεσματική από τη στρατηγική του Local History (διατήρηση τοπικού ιστορικού και ξεχωριστού πίνακα προτύπων).

4. **Global-History Predictors:** Φαίνεται να επιτυγχάνουν πολύ καλή απόδοση. Σύμφωνα με τα αποτελέσματα, υπερτερούν τόσο των Local-History predictors όσο και του N=4 predictor.

5. **Pentium-M και Alpha 21264 Predictors:** Ο Pentium-M predictor επέδειξε απόδοση συγκρίσιμη με τον *BTFNT*, ενώ ο *Alpha 21264* τοποθετήθηκε μεταξύ των Tournament predictors, καθώς και ο ίδιο είναι hybrid predictor.

6. **Tournament Predictors:** Οι υβριδικοί προγνώστες τύπου τουρνουά φαίνεται να επιτυγχάνουν καλή προς μέτρια απόδοση, τοποθετούμενοι γενικά μετά τους Global History και N-bit predictors.

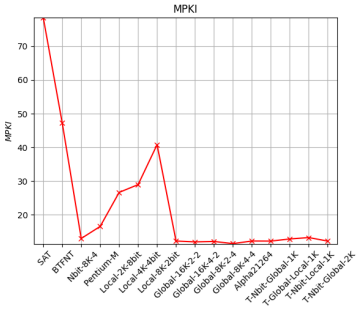
    - *Καλύτερος Tournament:* Μεταξύ των διαφορετικών διαμορφώσεων Tournament που δοκιμάστηκαν, η καλύτερη απόδοση παρατηρήθηκε για τον συνδυασμό ενός N-bit predictor (N=2, 8K entries) με έναν Global History predictor (PHT 8K entries), χρησιμοποιώντας τον meta-predictor με 1024 καταχωρήσεις.

Επιλογή Βέλτιστου Predictor Συνολικά:

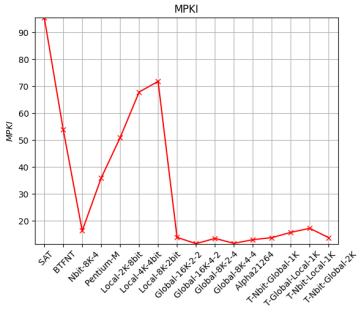
Με βάση το σχήμα 7, οι **Global History predictors** (και συγκεκριμένα η καλύτερη διαμόρφωση εξ αυτών, $BHR\_Length = 2, 4$, $PHT = 16K$, 4bit counters;) φαίνεται να προσφέρουν την υψηλότερη απόδοση.

Επομένως, ως βέλτιστος predictor από όσους εξετάστηκαν, θα επιλέγαμε τον **Global History με BHR=2, PHT=16K entries, 4-bit counters**. Οι Tournament και N-bit predictors αποτελούν μια καλή εναλλακτική, πλησιάζοντας την απόδοση των Global History, ενώ οι υπόλοιποι δυναμικοί predictors (Local History, Pentium-M, Alpha 21264) ακολουθούν στην κατάταξη απόδοσης.

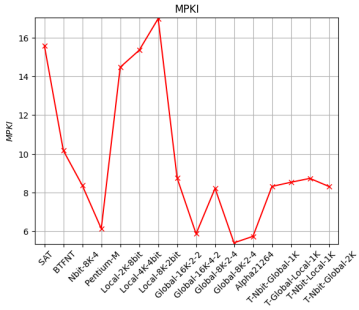
Figure 7: Results of the predictors comparison



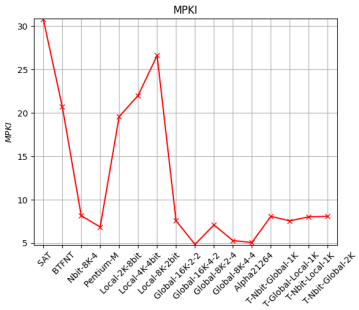
(a) 401.bzip2



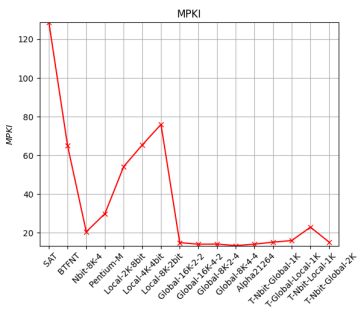
(b) 403.gcc



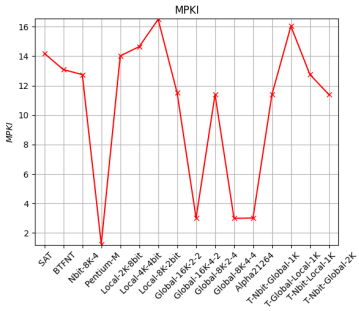
(c) 410.bwaves



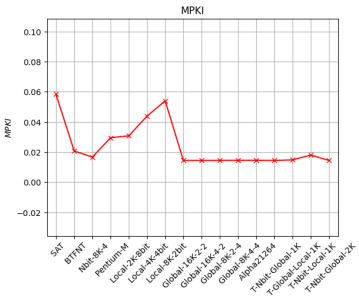
(d) 416.gamess



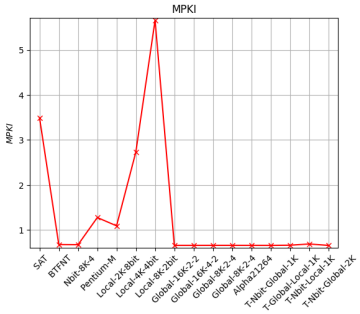
(e) 429.mcf



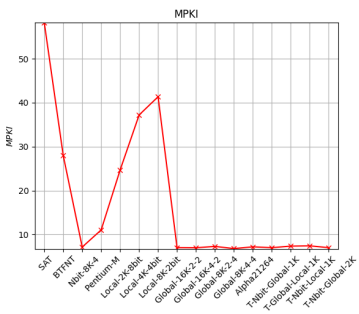
(f) 433.milc



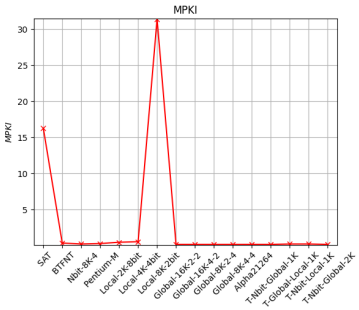
(g) 436.cactusADM



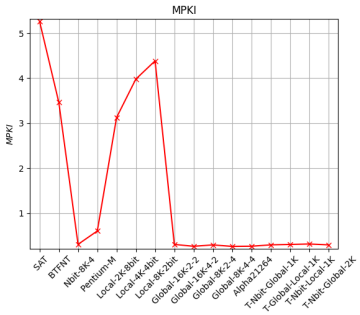
(h) 437.leslie3d



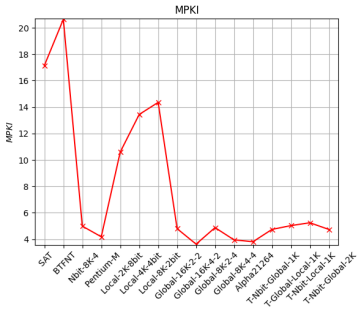
(i) 450.soplex



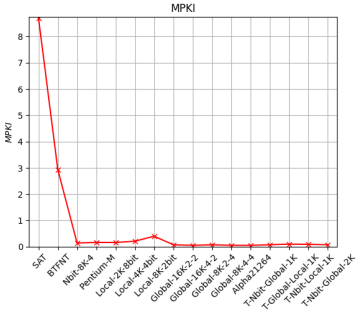
(j) 456.hmmer



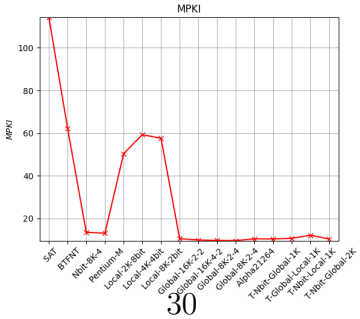
(k) 459.GemsFDTD



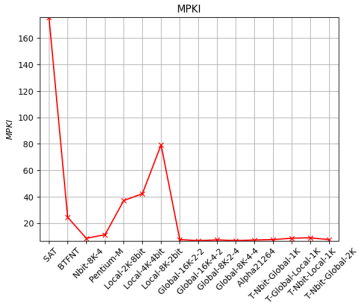
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

Train vs Ref Inputs (Άσκηση 5.7)

1 - Σκοπός και Μεθοδολογία

Οι προσομοιώσεις που πραγματοποιήθηκαν στα προηγούμενα ερωτήματα (5.3 - 5.6) χρησιμοποίησαν τα δεδομένα εισόδου αναφοράς (*ref inputs*). Αυτά τα inputs οδηγούν συνήθως σε μεγαλύτερες χρονικά και πιο αντιπροσωπευτικές εκτελέσεις των benchmarks. Ωστόσο, η χρήση των σημαντικά μικρότερων *train inputs* θα μπορούσε να μειώσει δραστικά τον χρόνο που απαιτείται για τις προσομοιώσεις.

Για να το δούμε αυτό, επαναλήφθηκαν οι προσομοιώσεις χρησιμοποιώντας τα *train inputs* για ένα υποσύνολο των predictors που εξετάστηκαν στο ερώτημα 5.6. Τα αποτελέσματα (MPKI) για τα *train inputs* παρουσιάζονται στο Σχήμα 8.

2 - Σύγκριση Αποτελεσμάτων και Συμπεράσματα

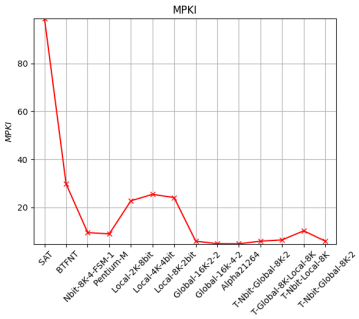
Η σύγκριση των αποτελεσμάτων που λήφθηκαν με τα *train inputs* (Σχήμα 8) με τα αντίστοιχα αποτελέσματα από τα *ref inputs* (Σχήμα 7) για το επιλεγμένο υποσύνολο των predictors οδηγεί στις εξής παρατηρήσεις:

- **Σχετική Κατάταξη:** Οι predictors που εμφάνισαν χαμηλότερο MPKI (καλύτερη απόδοση) με τα *ref inputs* τείνουν να διατηρούν την υπεροχή τους και με τα *train inputs*, και αντίστροφα.
- **Απόλυτες Τιμές MPKI:** Όπως και η σχετική κατάταξη έτσι και οι απόλυτες τιμές MPKI για έναν δεδομένο predictor διατηρούνται σταθερές μεταξύ των *train* και *ref inputs*.

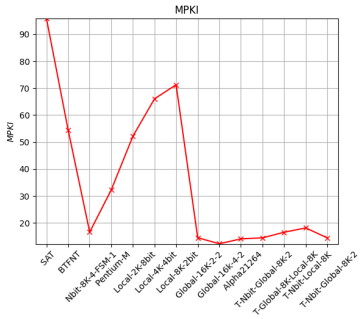
3 - Συμπεράσματα

Με βάση την παρατήρηση ότι η **σχετική κατάταξη** των εξεταζόμενων προγνωστών παραμένει ουσιαστικά η ίδια ανεξαρτήτως του αν χρησιμοποιήθηκαν τα *train* ή τα *ref inputs*, μπορούμε να συμπεράνουμε ότι για τους συγκεκριμένους predictors και benchmarks, θα είχαμε καταλήξει στα ίδια συμπεράσματα όσον αφορά την ανάδειξη των πιο αποδοτικών αρχιτεκτονικών πρόβλεψης, ακόμα κι αν είχαμε βασιστεί αποκλειστικά στις ταχύτερες προσομοιώσεις με τα *train inputs*.

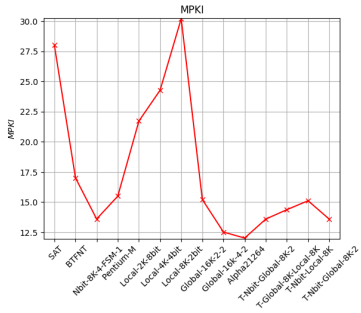
Ωστόσο, είναι σημαντικό να τονιστεί ότι τα *ref inputs* θεωρούνται πιο αντιπροσωπευτικά των πραγματικών φορτίων εργασίας και παραμένουν απαραίτητα για την τελική, ακριβή αξιολόγηση της απόδοσης ενός συγκεκριμένου σχεδιασμού predictor. Η ομοιότητα των συμπερασμάτων σε αυτή την περίπτωση δεν εγγυάται ότι το ίδιο θα ισχύει για όλες τις πιθανές αρχιτεκτονικές ή μετρικές απόδοσης.



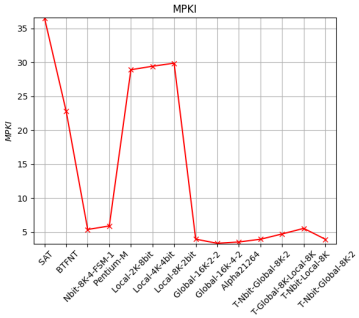
(a) 401.bzip2



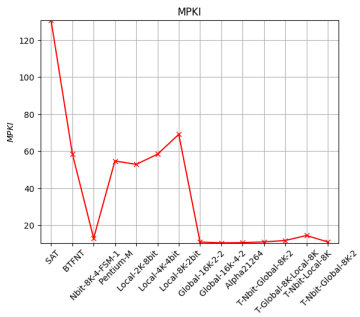
(b) 403.gcc



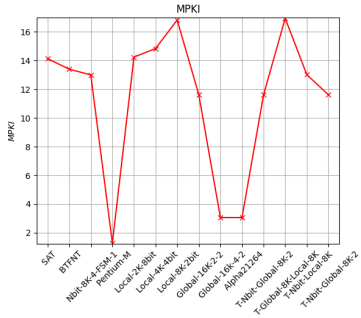
(c) 410.bwaves



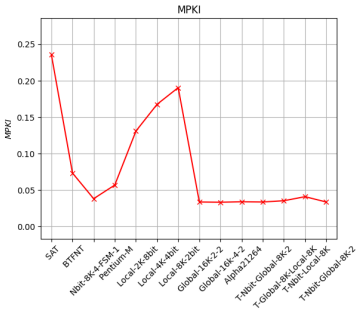
(d) 416.gamess



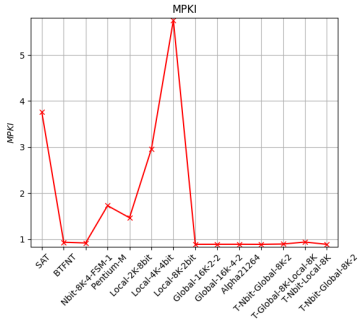
(e) 429.mcf



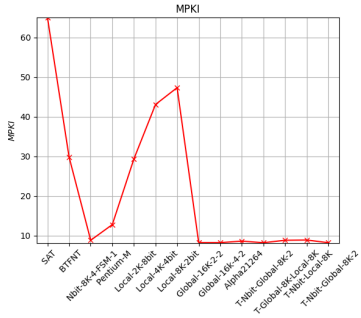
(f) 433.milc



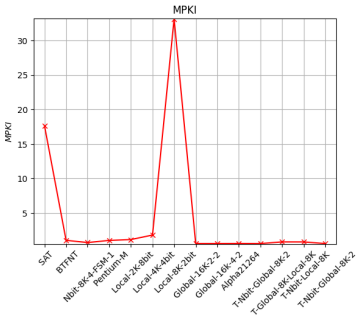
(g) 436.cactusADM



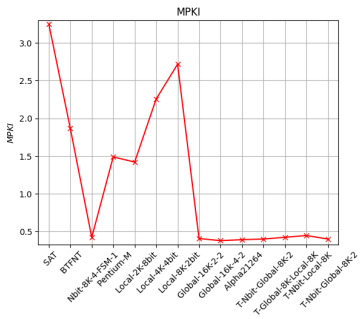
(h) 437.leslie3d



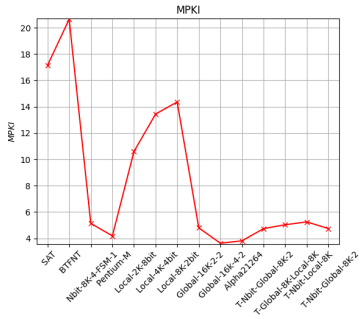
(i) 450.soplex



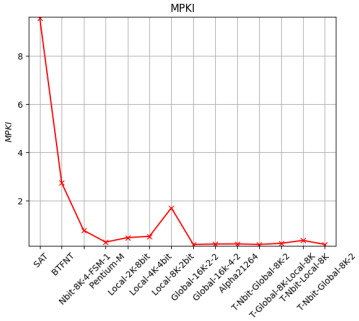
(j) 456.hmmer



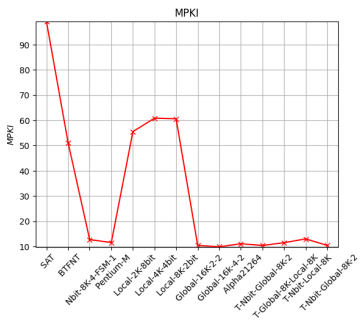
(k) 459.GemsFDTD



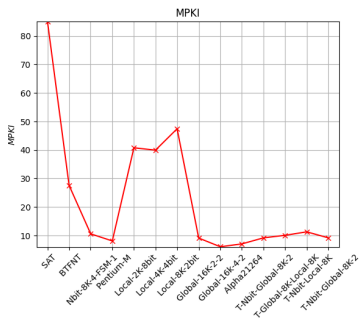
(l) 464.h264ref



(m) 470.lbm



(n) 471.omnetpp



(o) 483.xalancbmk

Figure 8: Predictor Comparison Train Inputs