2η ΆΣΚΗΣΗ ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Ενρίκα Ηλιάνα Ματζόρι ΑΜ:03120143

4. Πειραματική Αξιολόγηση

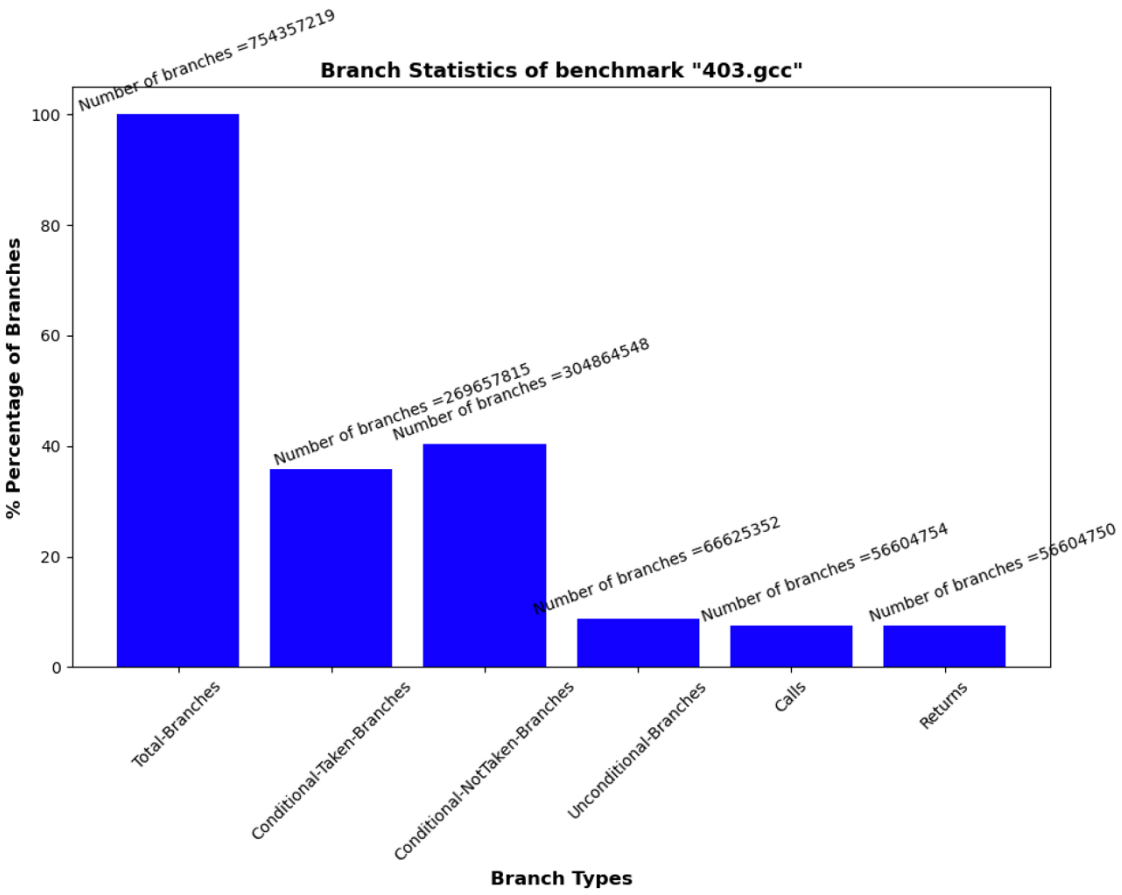
4.1 Μελέτη εντολών άλματος

Τρέχοντας το εξής script συλλέγουμε πληροφορίες για τα είδη και το πλήθος των εντολών άλματος που βρίσκουμε σε 14 διαφορετικά benchmarks.

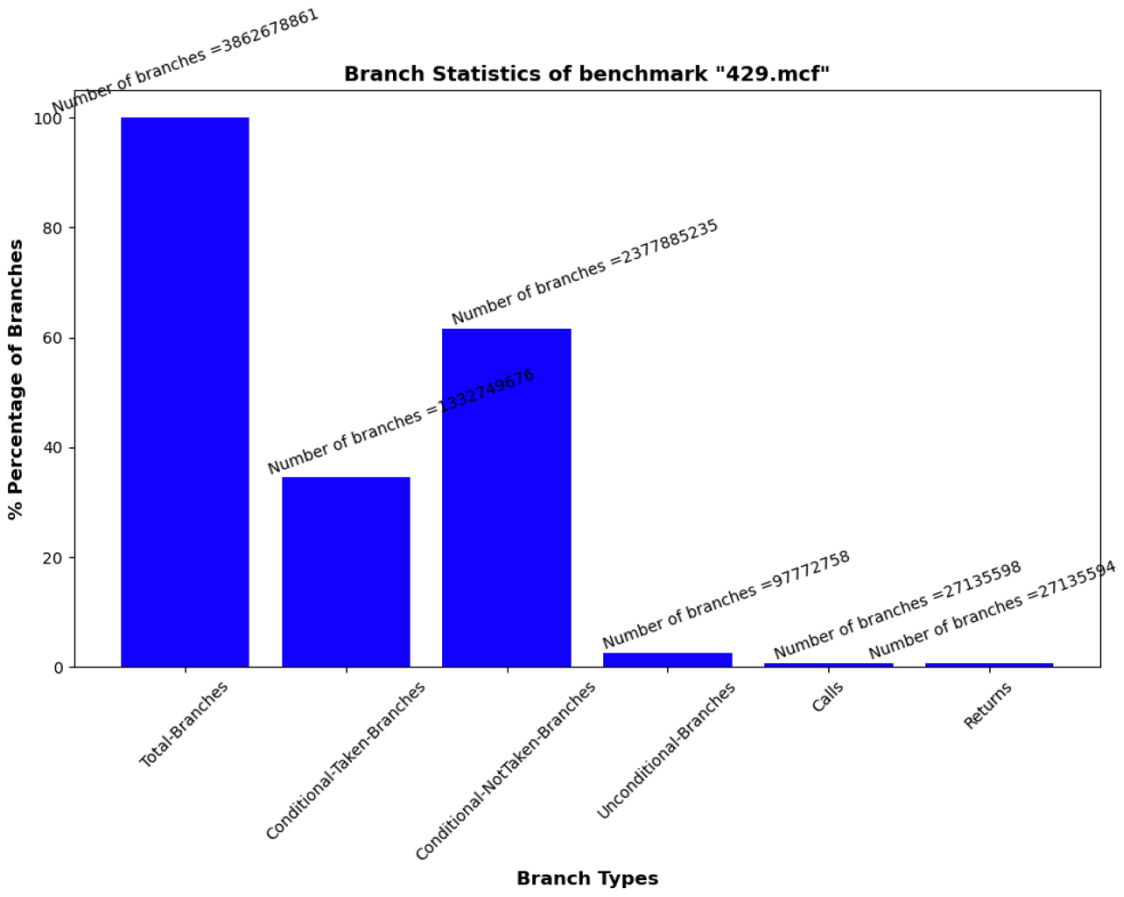
```
#!/bin/bash

## Execute this script in the helpcode directory.
## Example of usage: ./run_branch_predictors.sh 403.gcc
## Modify the following paths appropriately
## CAUTION: use only absolute paths below!!!
PIN_EXE=/home/iliana/Downloads/pin-3.30-98830-g1d7b601b3-gcc-linux/pin
PIN_TOOL=/home/iliana/Downloads/advcomparch-ex2-helpcode/pintool/obj-intel64/
cslab_branch_stats.so
outDir="/home/iliana/Downloads/advcomparch-ex2-helpcode/outputs_4.1"


benchmarks=("403.gcc" "436.cactusADM" "456.hmmer" "462.libquantum" "473.astar"
"429.mcf" "445.gobmk" "458.sjeng" "470.lbm" "483.xalancbmk" "434.zeusmp" "450.soplex"
"459.GemsFDTD" "471.omnetpp")
for BENCH in "${benchmarks[@]}"; do
        echo -e "$BENCH\n"
        cd spec_execs_train_inputs/$BENCH

        line=$(cat speccmds.cmd)
        stdout_file=$(echo $line | cut -d' ' -f2)
        stderr_file=$(echo $line | cut -d' ' -f4)
        cmd=$(echo $line | cut -d' ' -f5-)

        pinOutFile="$outDir/${BENCH}.cslab_branch_predictors.out"
        pin_cmd="$PIN_EXE -t $PIN_TOOL -o $pinOutFile -- $cmd 1> $stdout_file 2>
$stderr_file"
        echo "PIN_CMD: $pin_cmd"
        /bin/bash -c "time $pin_cmd"

        cd ../../
done
```

Συνοψίζουμε τις πληροφορίες που λαμβάνουμε αρχικά σε αρχεία κειμένου και στην συνέχεια τις αναπαριστούμε σε διαγράμματα όπου φαίνεται ο αριθμός των εντολών άλματος που εκτελέστηκαν και το ποσοστό αυτών που ανήκει στην κάθε κατηγορία. Για το κάθε benchmark λοιπόν έχουμε:

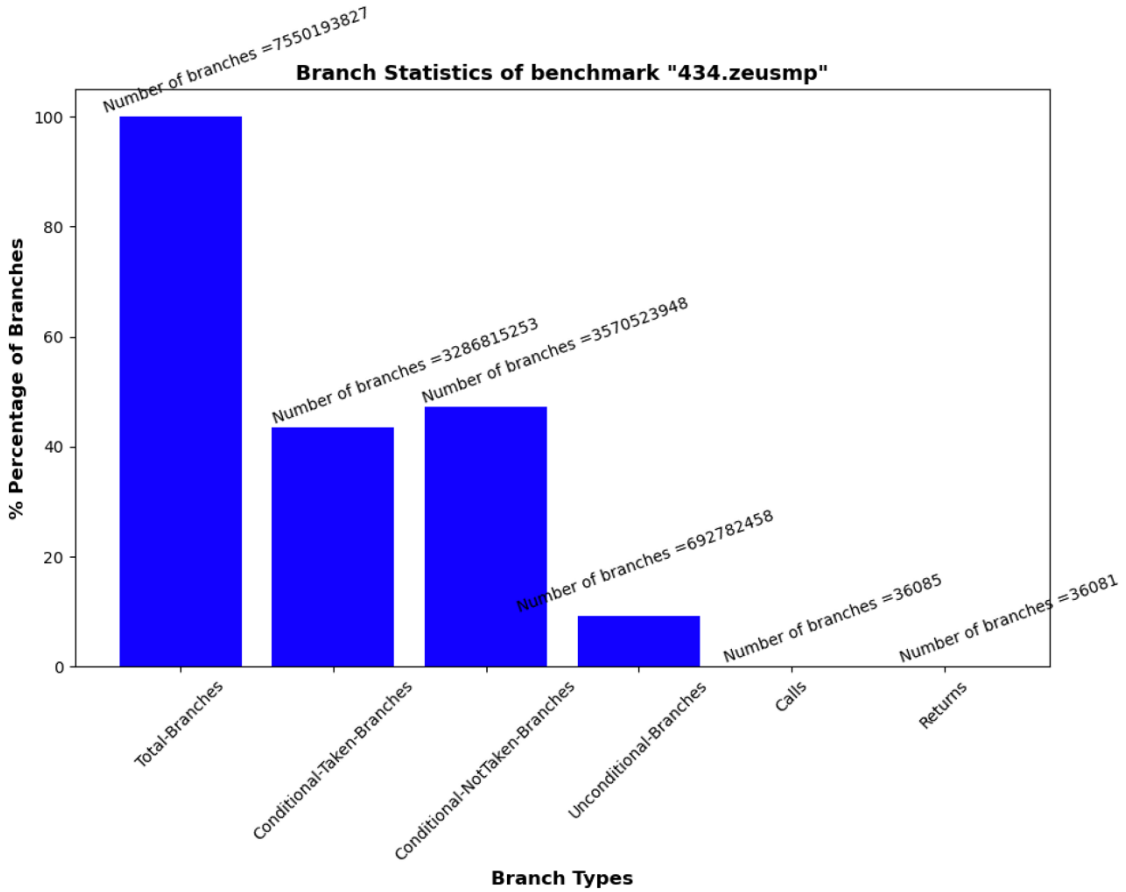
- Για το **403.gcc** έχουμε(ζητώ συγγνώμη για τα πλάγια labels αλλά προέκυψαν γιατί αλλιώς στις κορυφές των στηλών οι αριθμοί επειδή είναι πολύ μεγάλοι έπεφταν ο ένας πάνω στον άλλον ☹):



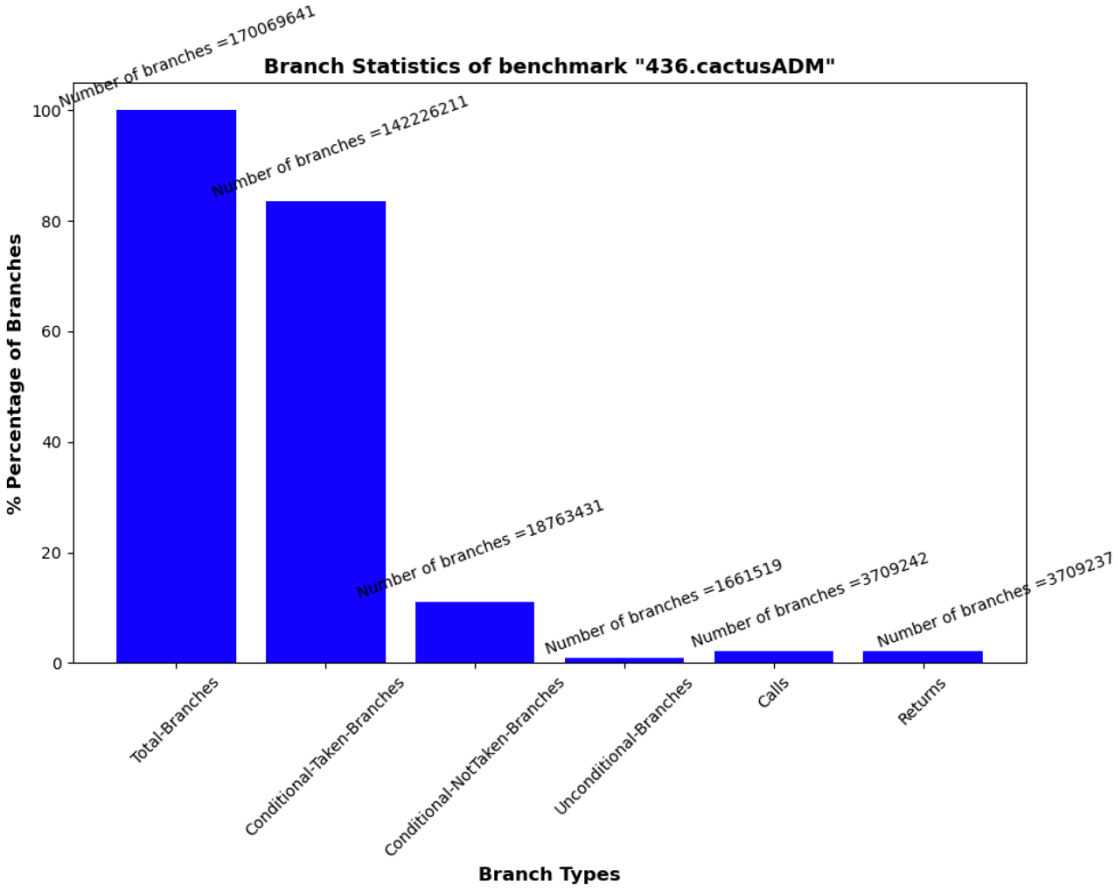
- Για το **429.mcf** έχουμε:



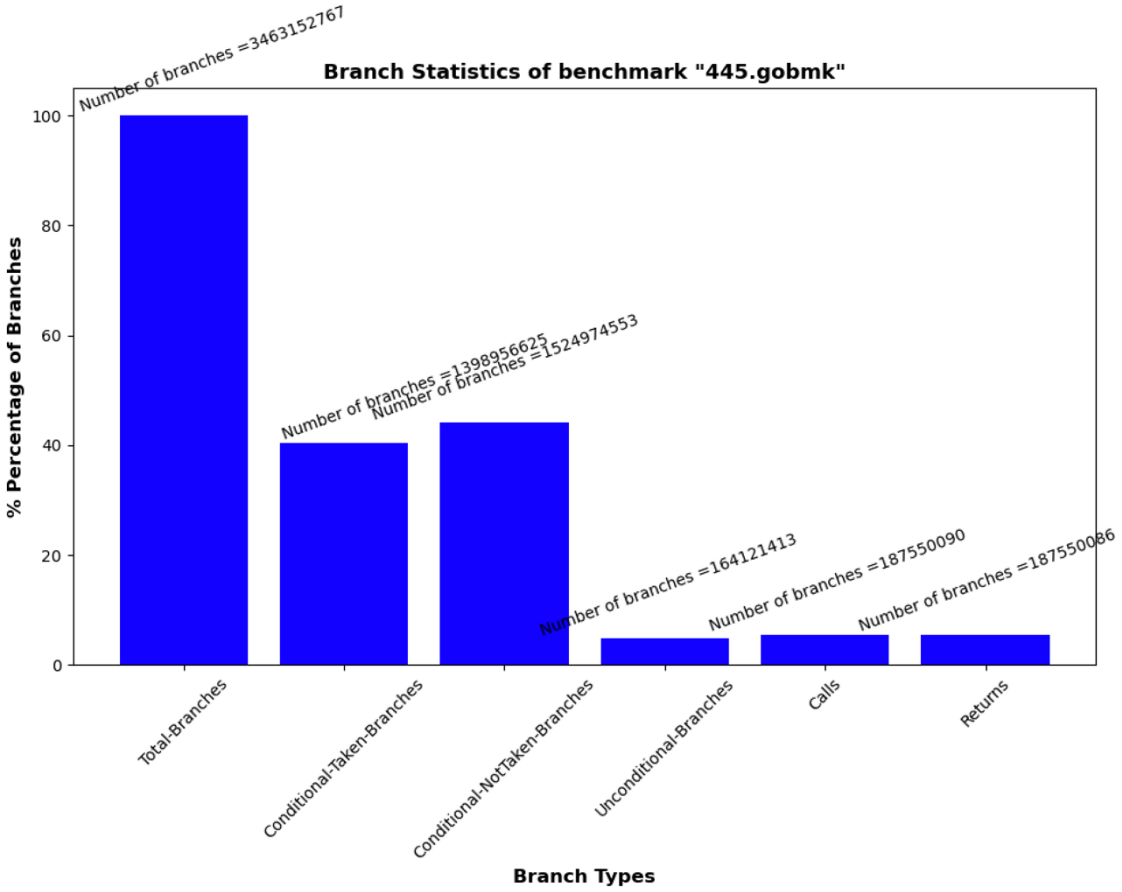
- Για το **434.zeusmp** έχουμε:



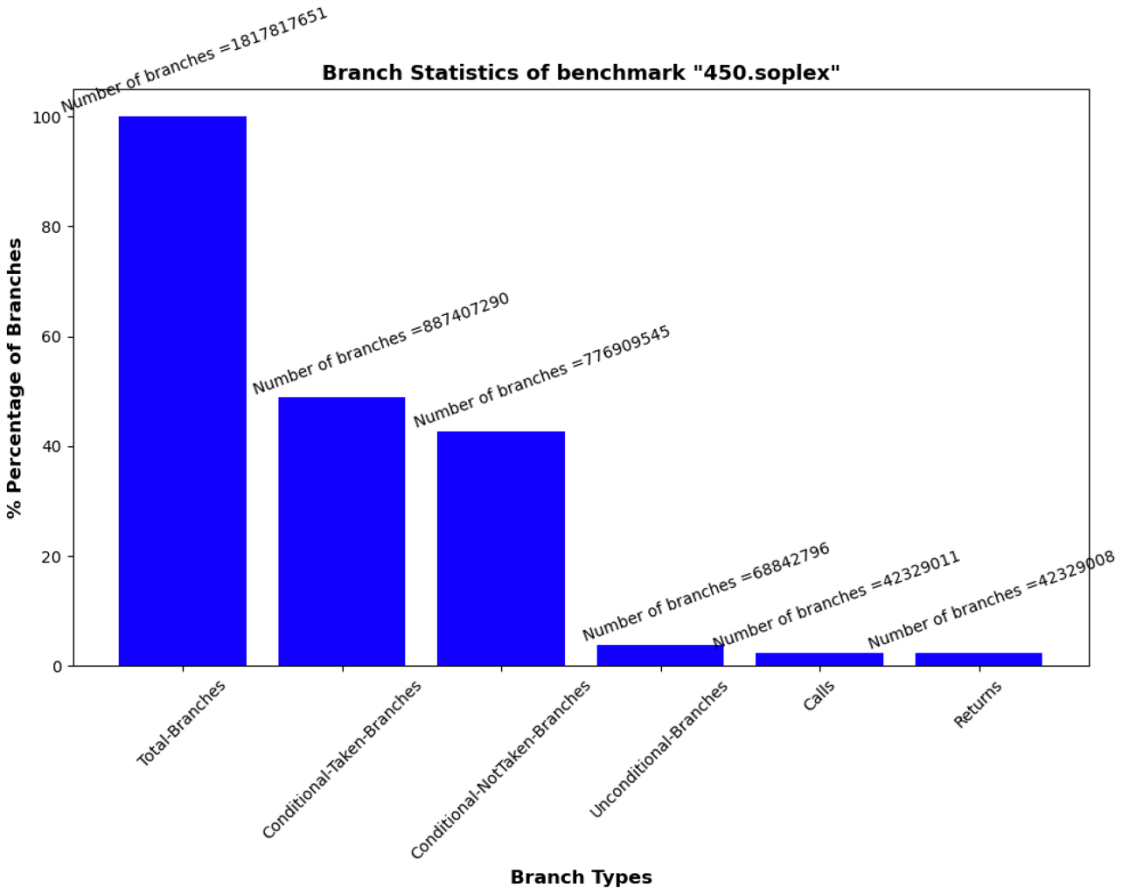
- Για το **436.cactusADM** έχουμε:



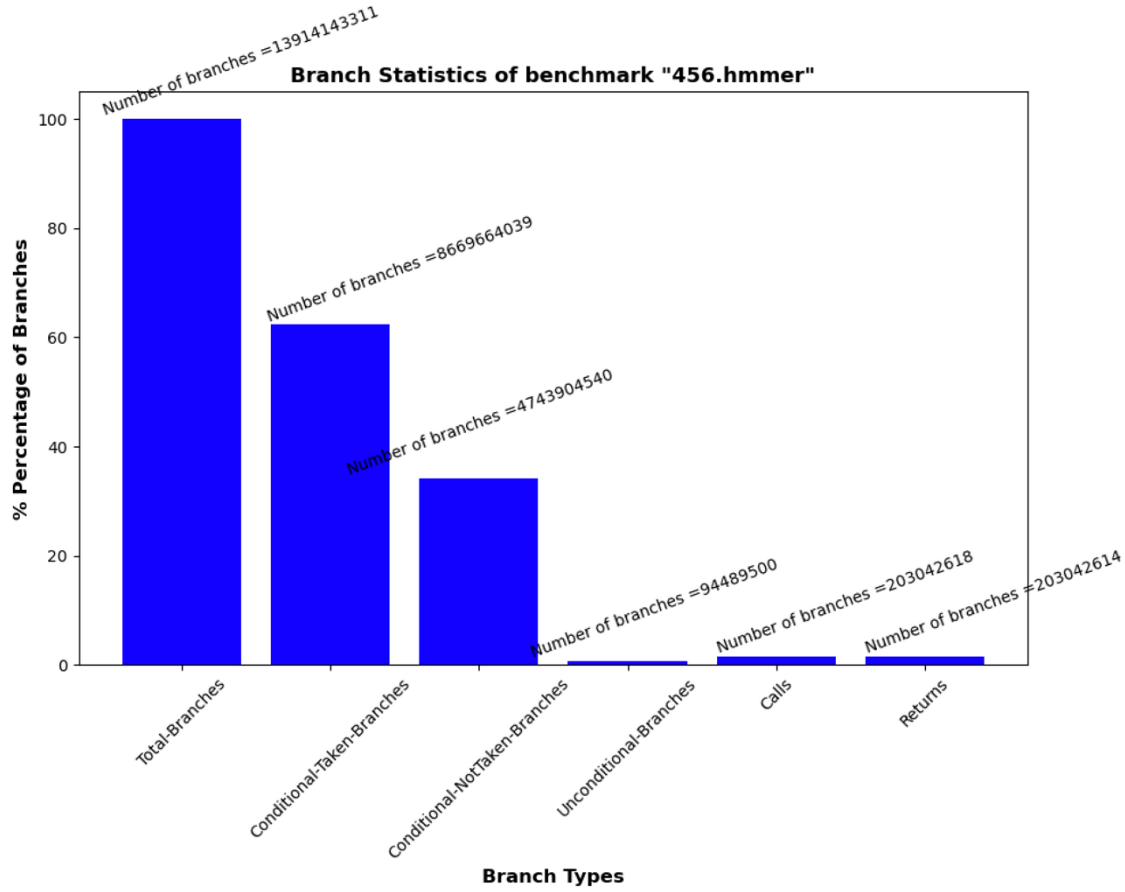
- Για το **445.gobmk** έχουμε:



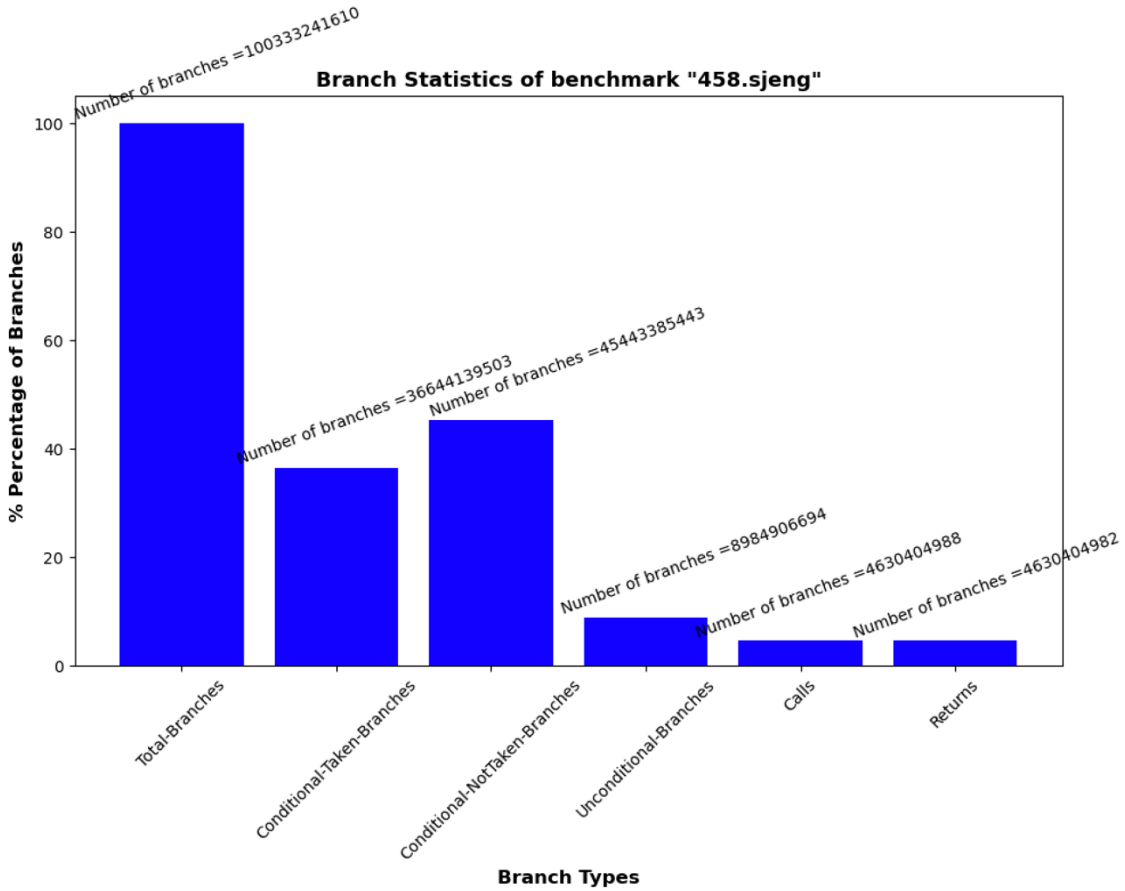
- Για το **450.soplex** έχουμε:



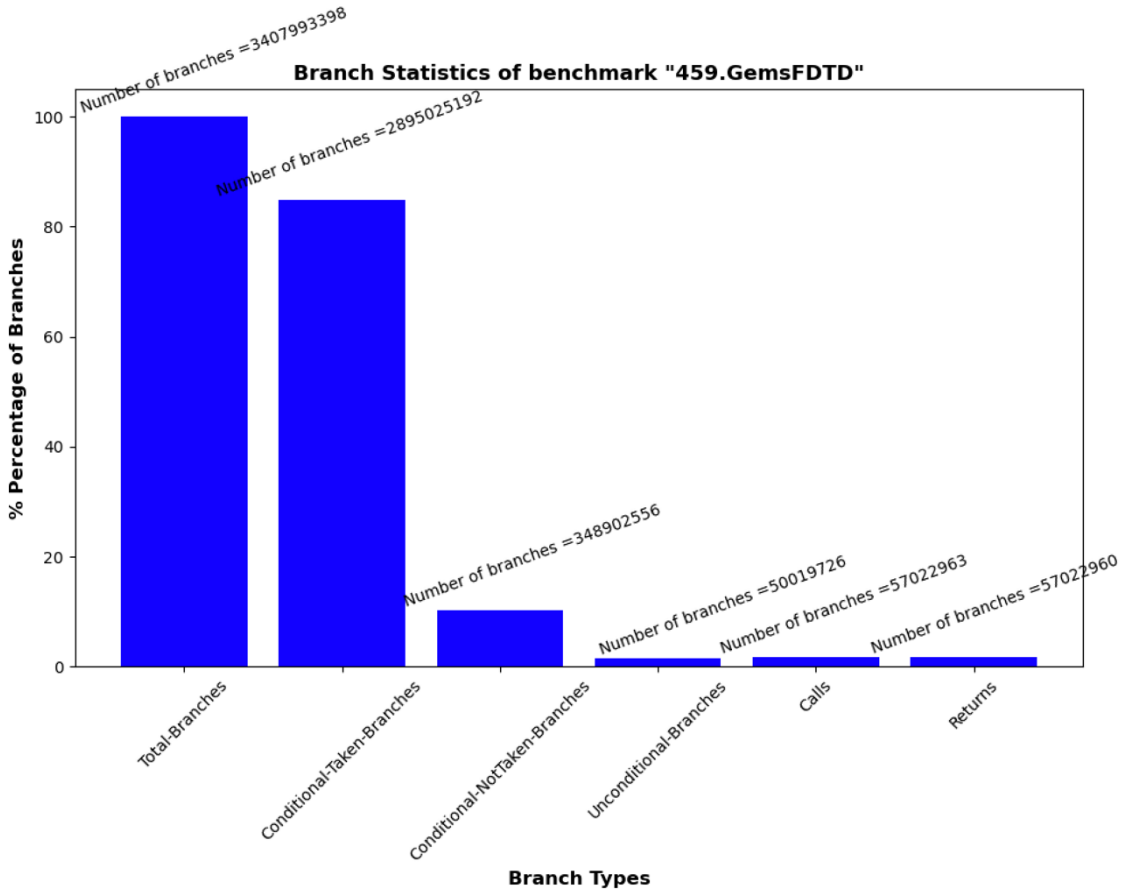
- Για το **456.hmmer** έχουμε:



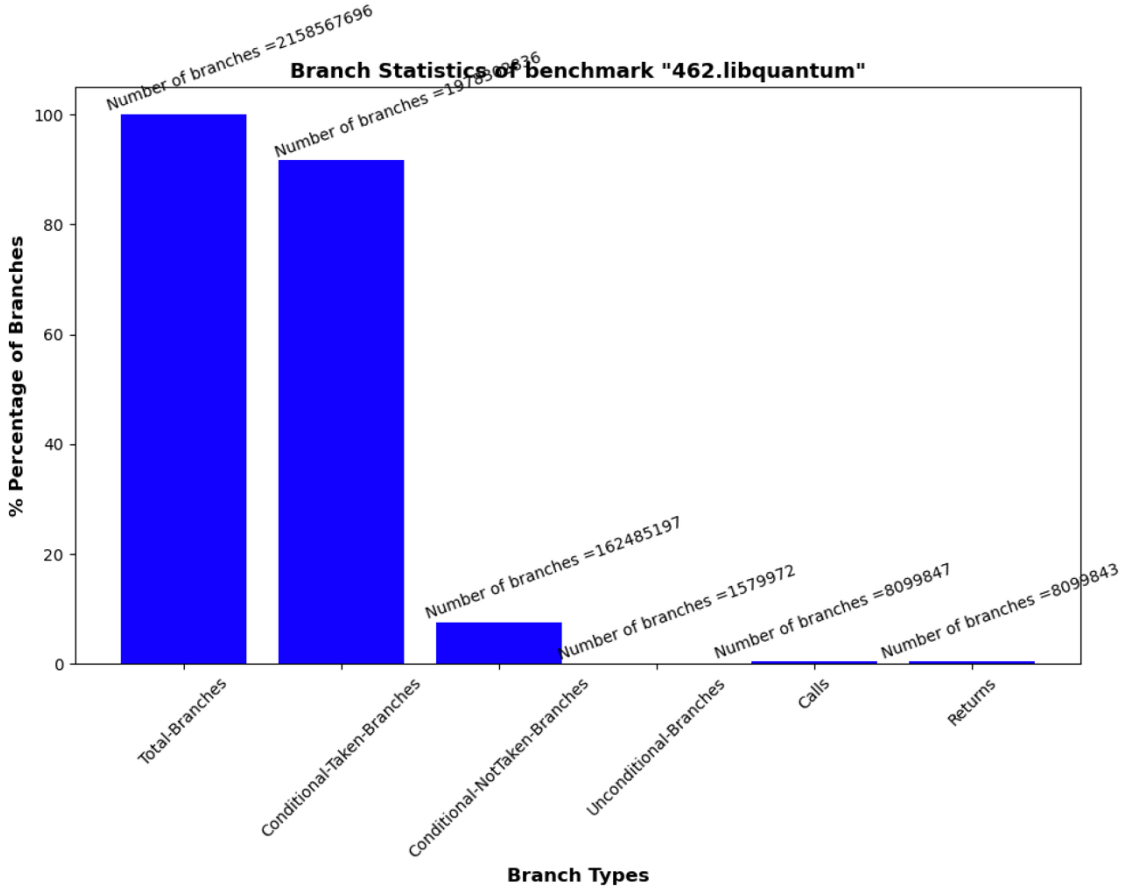
- Για το **458.sjeng** έχουμε:



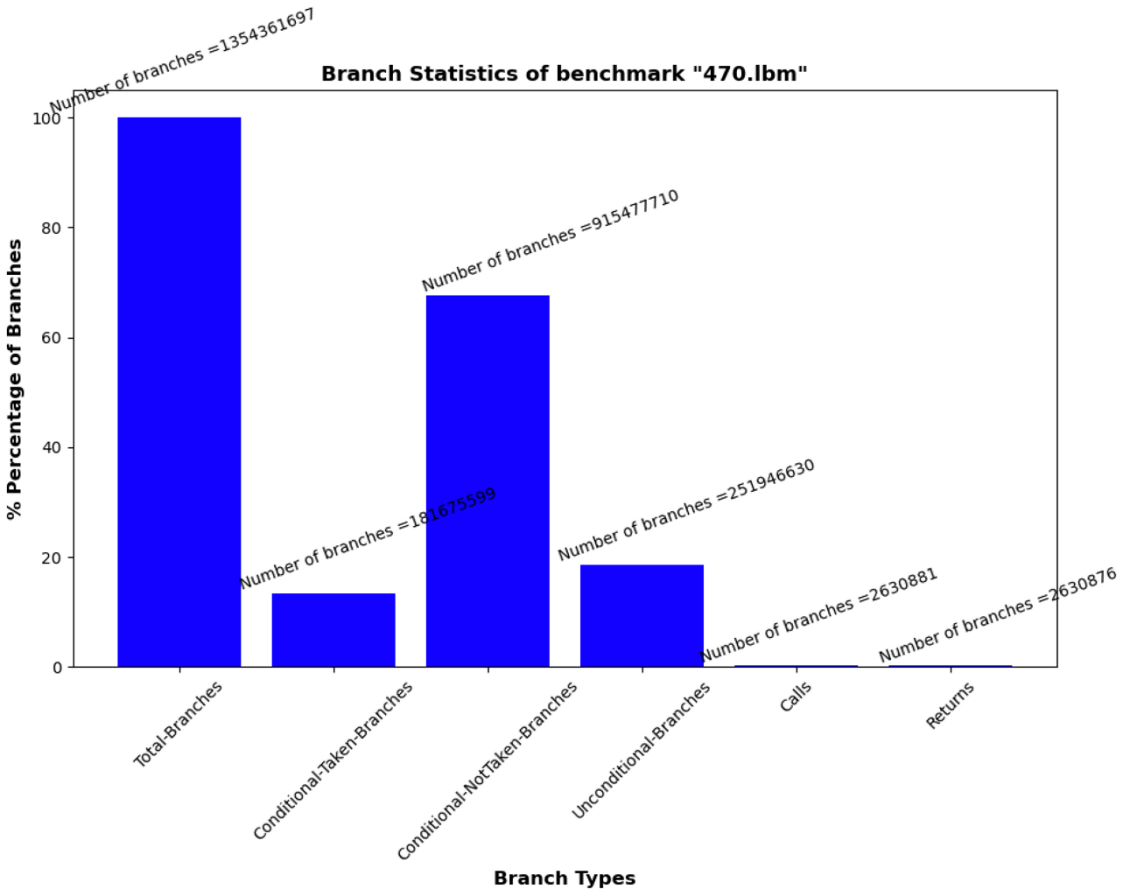
- Για το **459.GemsFDTD** έχουμε:



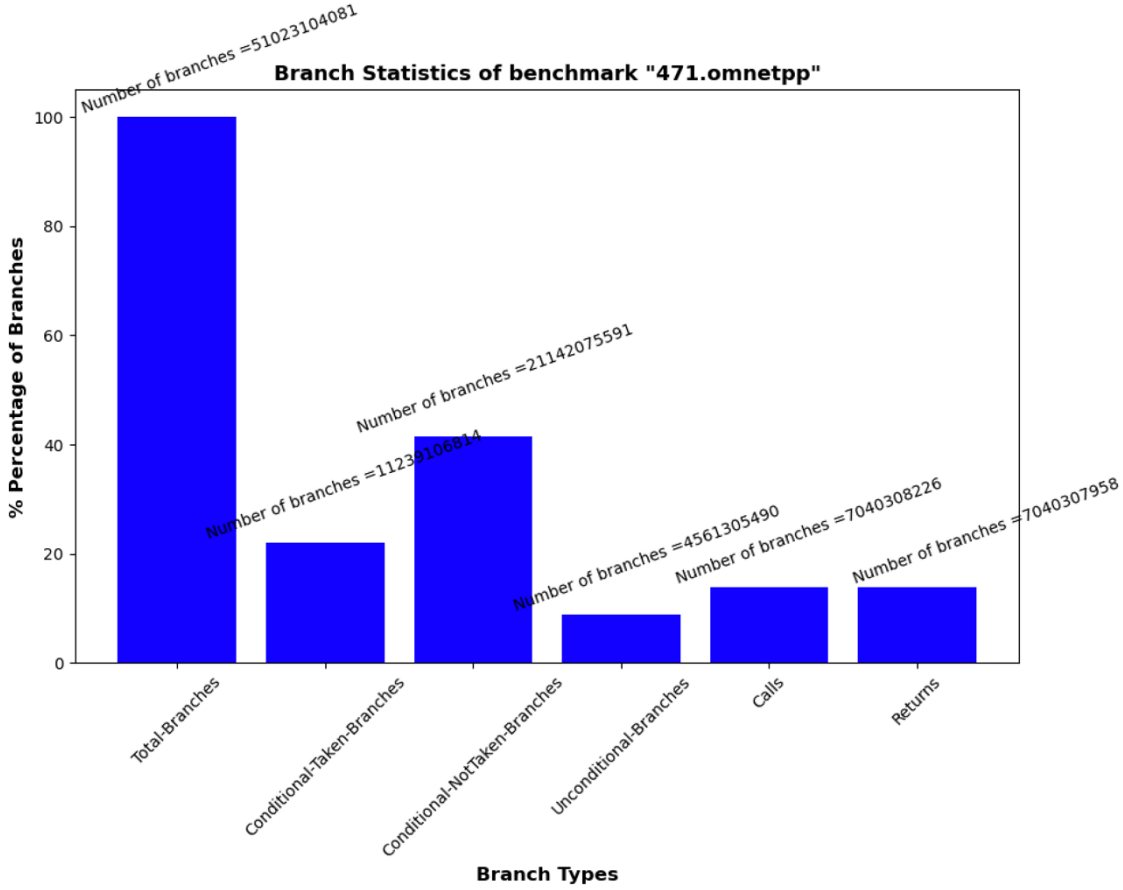
- Για το **462.libquantum** έχουμε:



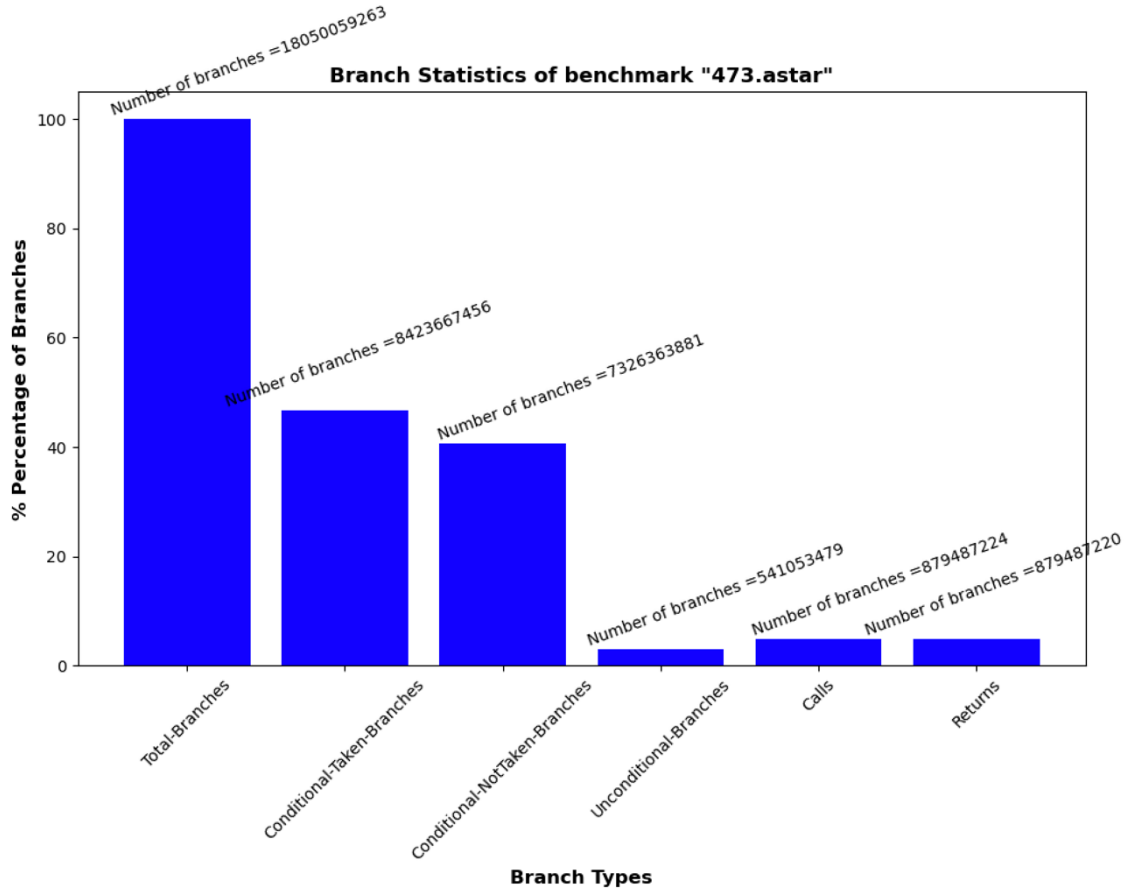
- Για το **470.lbm** έχουμε:



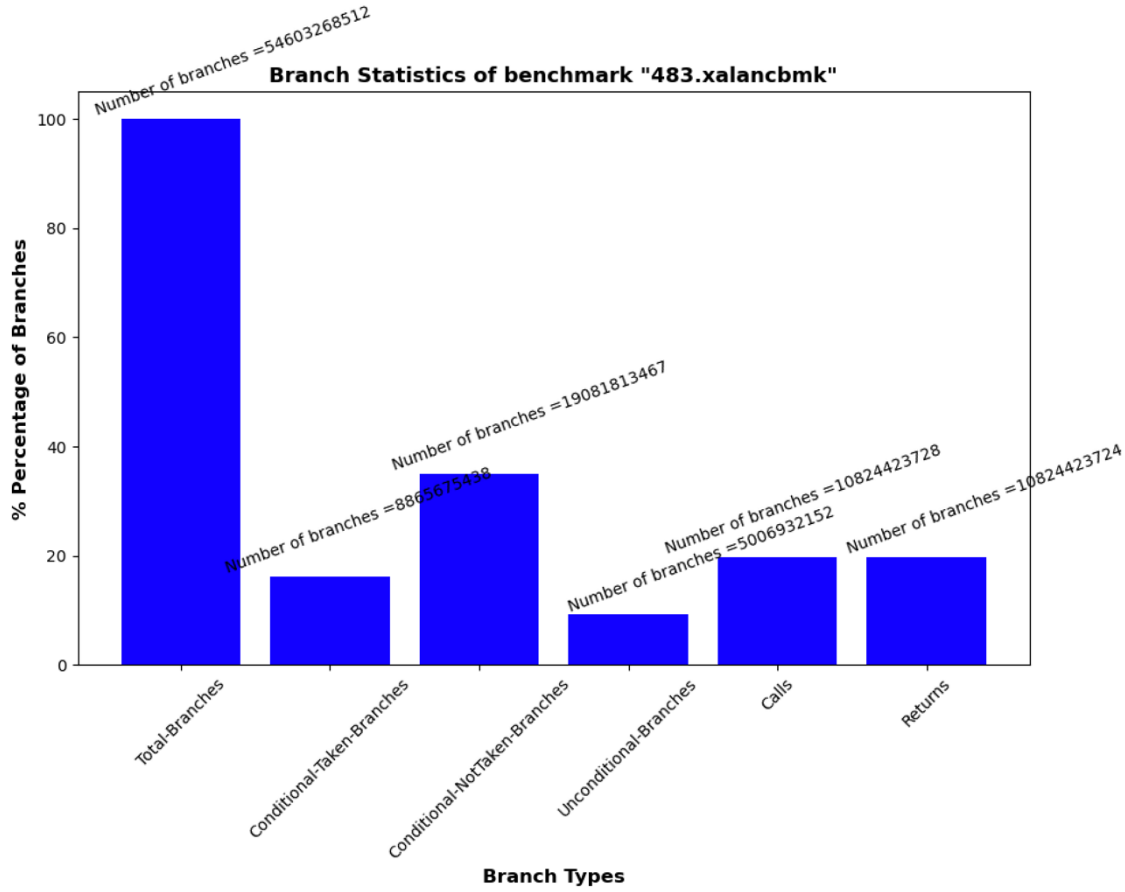
- Για το **471.omnetpp** έχουμε:



- Για το **473.astar** έχουμε:



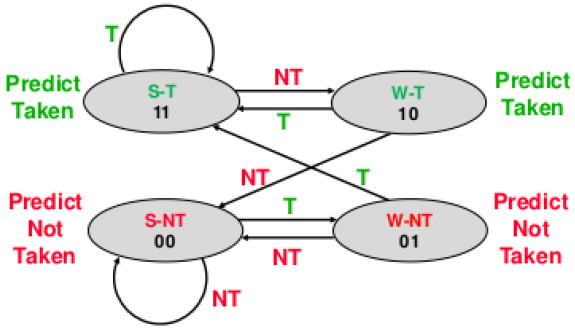
- Για το **483.xalancbmk** έχουμε:



Από τα παραπάνω στατιστικά για τα benchmarks αντιλαμβανόμαστε ότι το πλήθος των εντολών άλματος είναι μεγάλο φτάνοντας ακόμα και δεκαψήφιους αριθμούς. Υπάρχουν βέβαια και benchmarks τα οποία έχουν σημαντικά λιγότερες εντολές άλματος απ' τα υπόλοιπα, όπως το 436.cactusADM. Ωστόσο απ' τα αναλυτικά text αρχεία με τις πληροφορίες για το εκάστοτε benchmark βλέπουμε ότι και το πλήθος των συνολικών εντολών σε αυτά τα μετροπρογράμματα είναι πολύ μειωμένο σε σχέση με τα υπόλοιπα. Σε ότι αφορά τις κατηγορίες των αλμάτων, παρατηρούμε ότι σχεδόν πάντα τα περισσότερα είναι όσα ανήκουν στις κατηγορίες Conditional Taken και Conditional Not Taken, ενώ αρκετά λιγότερα είναι όσα ανήκουν στις υπόλοιπες κατηγορίες, αγγίζοντας τις περισσότερες φορές αθροιστικά το 10% των συνολικών εντολών άλματος.

4.2 Μελέτη των N-bit predictors

(I) Ζητούμενο της συγκεκριμένης άσκησης είναι να μελετηθεί η απόδοση των n-bits predictors για όλα τα benchmarks και για διαφορετικές τιμές του N, διατηρώντας σταθερό τον αριθμό των entries. Πέρα απ'τους N-bit predictors, υλοποιείται και ένας άλλος predictor του οποίου το FSM είναι το εξής:



Υλοποιούμε τον εξής predictor(που στις γραφικές αναγράφεται σαν 2bit-predictor) στο αρχείο branch_predictor.h:

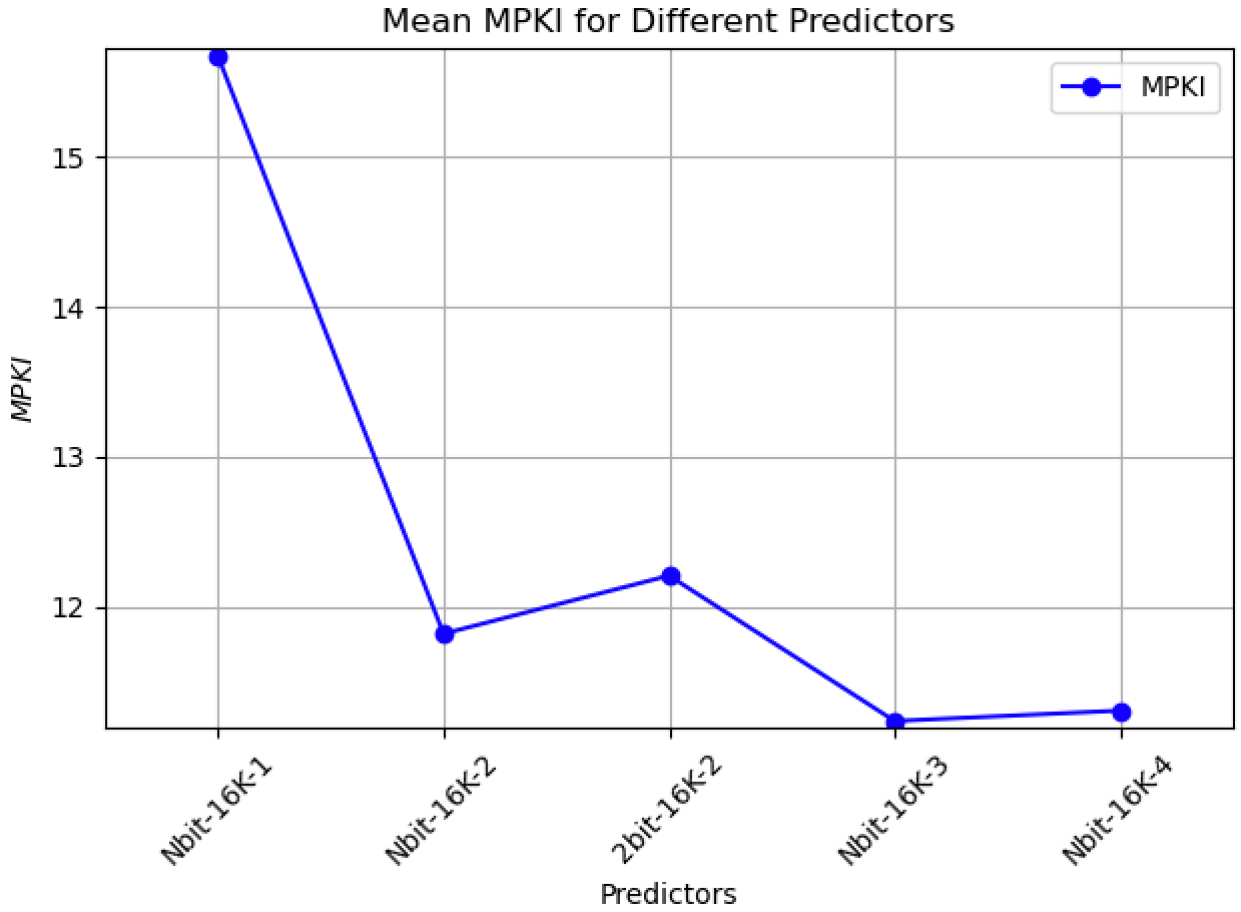
```
class TwobitPredictor : public BranchPredictor
{
public:
    TwobitPredictor(unsigned index_bits_, unsigned cntr_bits_)
        : BranchPredictor(), index_bits(index_bits_), cntr_bits(cntr_bits_) {
        table_entries = 1 << index_bits; //-->2^(index_bits)
        TABLE = new unsigned long long[table_entries];
        memset(TABLE, 0, table_entries * sizeof(*TABLE));

        COUNTER_MAX = (1 << cntr_bits) - 1;
    };
    ~TwobitPredictor() { delete TABLE; };

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

    virtual void update(bool predicted, bool actual, ADDRINT ip, ADDRINT target) {
        unsigned int ip_table_index = ip % table_entries;
        if (actual) {
            //if we are on 10 and branch is taken then we go to 11
            if (TABLE[ip_table_index] == 1)
                TABLE[ip_table_index] = 3;
            else if(TABLE[ip_table_index] < COUNTER_MAX){
                TABLE[ip_table_index]++; //we get closer to predicting "taken"
            }

        } else { //we follow the FSM
            if (TABLE[ip_table_index] == 2)
                TABLE[ip_table_index] = 0;
            else if(TABLE[ip_table_index] > 0){
                TABLE[ip_table_index]--;
            }
        }

        updateCounters(predicted, actual);
    };

    virtual string getName() {
        std::ostringstream stream;
        stream << "2bit-" << pow(2.0,double(index_bits)) / 1024.0 << "K-" << cntr_bits;
        return stream.str();
    }

private:
    unsigned int index_bits, cntr_bits;
    unsigned int COUNTER_MAX;

    /* Make this unsigned long long so as to support big numbers of cntr_bits. */
    unsigned long long *TABLE;
    unsigned int table_entries;
};
```

Προκειμένου να συγκρίνουμε τους predictors για τα διάφορα benchmarks, για κάθε είδος predictor, θα πάρουμε τον αριθμητικό μέσο όρο των direction MPKI (Mispredictions Per Thousand Instructions), για όλα τα benchmarks και θα τα αναπαραστήσουμε σε ένα διάγραμμα. Ο υπολογισμός της μέσης αυτής τιμής γίνεται με τον εξής κώδικα:

mean_MPKI.sh
~/Downloads/advcomparch-ex2-helpcode

```
#!/bin/bash

Outputs="/home/iliana/Downloads/advcomparch-ex2-helpcode/outputs_4.2"
Results="/home/iliana/Downloads/advcomparch-ex2-helpcode/mean_res_4_2"

# List of benchmarks
benchmarks=("403.gcc" "436.cactusADM" "456.hmmer" "462.libquantum" "473.astar" "429.mcf" "445.gobmk" "458.sjeng" "470.lbm"
 "483.xalancbmk" "434.zeusmp" "450.soplex" "459.GemsFDTD" "471.omnetpp")



predictors=("Nbit-16K-1" "2bit-16K-2" "Nbit-16K-2" "Nbit-16K-3" "Nbit-16K-4")
for predictor in "${predictors[@]}"; do
    MPKI_sum=0
    files_count=0
    # Create the output file
    out_file="${Results}/${predictor}_mean_MPKI.txt"
    # Iterate through each benchmark file
    for benchmark in "${benchmarks[@]}"; do
        pattern="${benchmark}.cslab_4_2b_branch_predictors.out"
        echo "Pattern: $pattern"
        # Iterate over files in the directory
        for file in "${Outputs}"/*; do
            # Check if the filename matches the pattern
            if [[ "$file" =~ $pattern ]]; then
                total_instructions=$(grep "Total Instructions:" "$file" | awk '{print $3}')
                # If it matches, take the information we need reading from the file
                incorrect_branches=$(grep "$predictor" "$file" | awk '{print $3}')
                # Calculate MPKI
                MPKI=$(echo "scale=6; (${incorrect_branches} * 1000) / ${total_instructions}" | bc)
                MPKI_sum=$(echo "scale=6; ${MPKI_sum} + ${MPKI}" | bc)
                # Increasing of files_count. At the end it should be 14 as we have 14 different benchmarks
                ((files_count++))
            fi
        done
    done
    mean_MPKI=$(echo "scale=6; ${MPKI_sum} / ${files_count}" | bc)
    # Store the results in the output file created
    echo "For predictor ${predictor} and for all ${files_count} benchmarks we found that: " >> "$out_file"
    echo "Mean MPKI: ${mean_MPKI}" >> "$out_file"
done
```

Αναπαριστώντας τις τιμές που υπολογίσαμε για τον κάθε predictor σε ένα διάγραμμα έχουμε:



Παρατηρούμε ότι η επίδοση αυξάνεται,αφού το dMPKI φθίνει, με την αύξηση του πλήθους των bits. Ωστόσο την μεγαλύτερη διαφορά παρατηρούμε από το 1 bit σε σύγκριση με όλα τα μεγαλύτερα. Αξίζει να σημειώσουμε ότι στην περίπτωση των 3-bit-predictor, 4-bit-predictor υπάρχει μία ελαφριά,σχεδόν ανεπαίσθητη, μείωση της απόδοσης. Αναφορικά με το δικό μας FSM(2bit-16K-2), παρατηρούμε ότι μας δίνει καλύτερη απόδοση μόνο σε σύγκριση με τον N-bit-predictor για N=1.
Κρατώντας ως κριτήριο την επίδοση, απ'το διάγραμμα είναι φανερό ότι η καλύτερη επιλογή είναι ο 16K 3-bit Predictor, όπου δηλαδή το MPKI ελαχιστοποιείται. Αξίζει να σημειώσουμε βέβαια ότι όταν τα entries μένουν σταθερά και το N αυξάνεται, οι απαιτήσεις σε hardware επίσης αυξάνονται. Επομένως αν μας ενδιαφέρει αυτό πολύ, θυσιάζοντας λίγο από την συνολική επίδοση αλλά χρησιμοποιώντας λιγότερο hardware, θα μπορούσαμε να επιλέξουμε και τον Nbit Predictor για N=2.

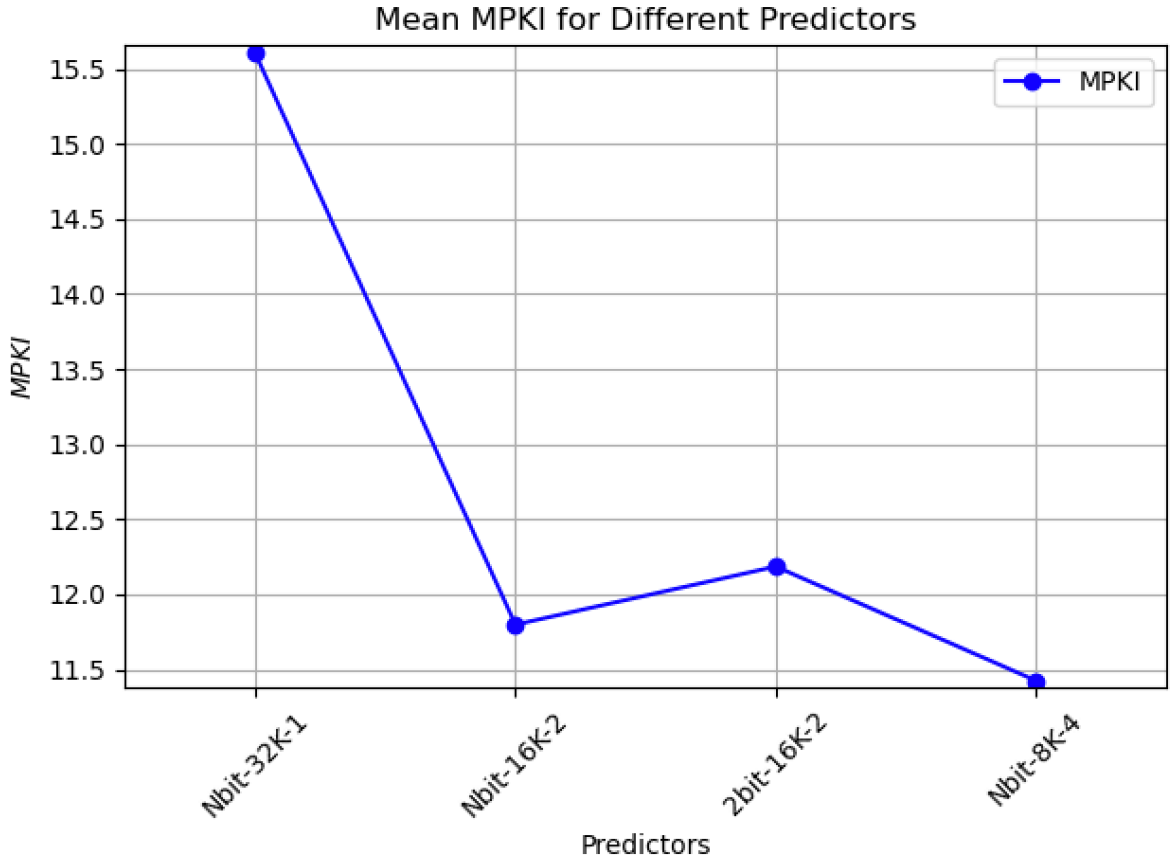
(II) Επαναλαμβάνουμε την διαδικασία που είδαμε παραπάνω για την μελέτη της απόδοσης των διαφορετικών benchmarks αλλά αυτή τη φορά θα κρατήσουμε σταθερό το μέγεθος του hardware που θα χρησιμοποιήσουμε και ίσο με 32K bits, μεταβάλλοντας τον αριθμό των entries στην εκάστοτε περίπτωση. Ο αριθμός των entries αλλάζει ανάλογα τον predictor που θα χρησιμοποιήσουμε κάθε φορά, ανάλογα δηλαδή το N. Οπότε για κάθε τιμή του N, προκειμένου το μέγεθος να μείνει σταθερό, θα έχουμε:

- Για N = 1 : entries = 32K/1 = 32K
- Για N = 2 : entries = 32K/2 = 16K
- Για N = 4 : entries = 32K/4 = 8K

Προκειμένου να δίνουμε το σωστό argument στις συναρτήσεις που ορίζουν τους predictors, μεταβάλλουμε τον κώδικα στο αρχείο cslab_branch.cpp ως εξής:

```
VOID InitPredictors()
{
    int entries=0;
    for (int i=1; i <= 4; i++) {
        //we want to maintain fixed the size of the table, so we change entries
        entries = log2(32 * 1024 / i);
        // for N=2 we want our FSM
        if(i==2){
                TwobitPredictor *twobitPred = new TwobitPredictor(entries, i);
                branch_predictors.push_back(twobitPred);
        }
        if(i != 3){
                NbitPredictor *nbitPred = new NbitPredictor(entries, i);
                branch_predictors.push_back(nbitPred);
        }
    }
```

Εκτελώντας τα benchmarks υπολογίζουμε όπως και πριν τους μέσους όρους των MPKI για κάθε είδος predictor και το αναπαριστούμε ως εξής:



Όπως και στο προηγούμενο ερώτημα, το MPKI φθίνει όσο αυξάνεται ο αριθμός του N ενώ το FSM που υλοποιήσαμε πετυχαίνει καλύτερη απόδοση μόνο συγκριτικα με τον 1 bit Predictor.
Παρατηρούμε επίσης ότι την καλύτερη απόδοση, δηλαδή την ελάχιστη τιμή του MPKI έχουμε στην περίπτωση του4bit predictor με 8K entries. Κρατώντας πάλι ως κριτήριο μας την απόδοση, μπορούμε να αποφανθούμε ότι καλύτερη επιλογή Predictor είναι ο 4bit predictor με 8K entries ακολουθούμενος απ' τον 2 bit predictor με 16K enties.

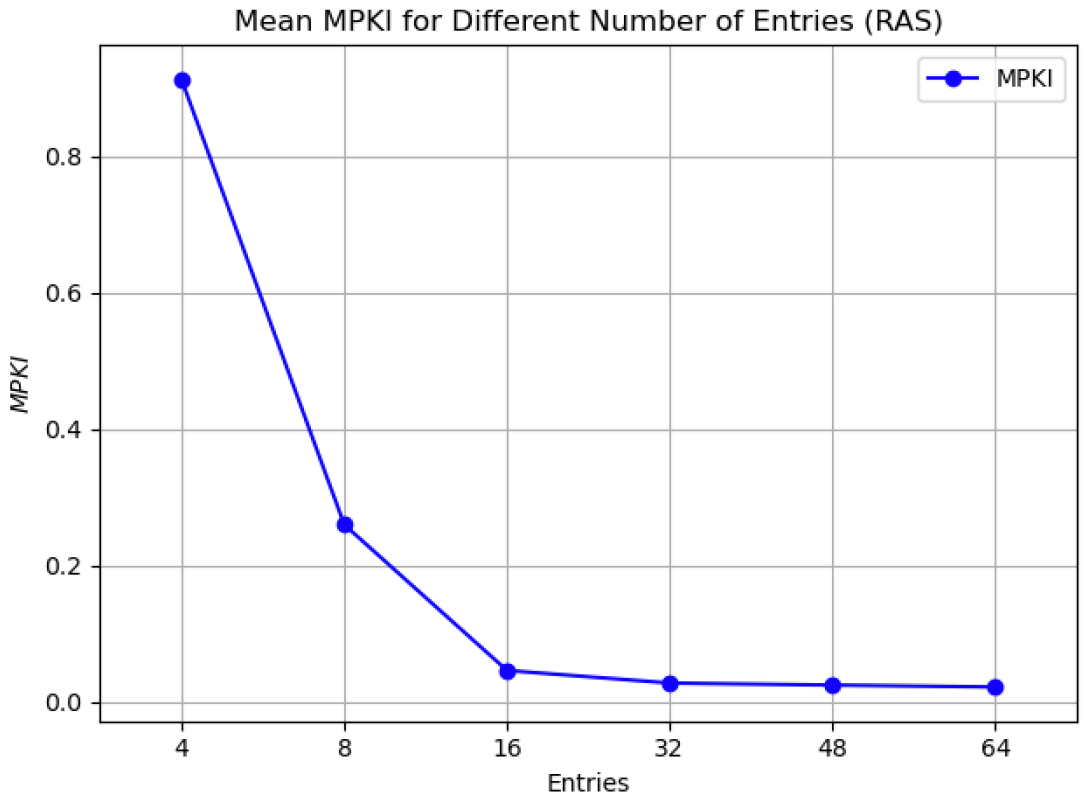
4.4 Μελέτη του RAS

Για διαφορετικές τιμές εγγραφών(entries) στην RAS στα διαφορετικά benchmarks επαναλαμβάνουμε την διαδικασία υπολογισμού των διαφόρων MPKI και εξαγωγής του μέσου όρου για όλα τα benchmarks για κάθε διαφορετική τιμή των entries προκειμένου να γίνει η μελέτη της επίδοσης στην κάθε περίπτωση. Για τον υπολογισμό του μέσου όρου του MPKI για τις διάφορες τιμές των entries εκτελέσαμε τον κώδικα:

Open ras_MPKI.sh ~/Downloads/advcomparch-ex2-helpcode Save

```
#!/bin/bash

Outputs="/home/iliana/Downloads/advcomparch-ex2-helpcode/outputs_4.4"
Results="/home/iliana/Downloads/advcomparch-ex2-helpcode/mean_res_4_4"

# List of benchmarks
benchmarks=("403.gcc" "436.cactusADM" "456.hmmer" "462.libquantum" "473.astar" "429.mcf" "445.gobmk" "458.sjeng" "470.lbm"
 "483.xalancbmk" "434.zeusmp" "450.soplex" "459.GemsFDTD" "471.omnetpp")


entries=("4" "8" "16" "32" "48" "64")
for entrie in "${entries[@]}"; do
    MPKI_sum=0
    files_count=0
    # Create the output file
    out_file="${Results}/${entrie}_mean_MPKI.txt"
    # Iterate through each benchmark file
    for benchmark in "${benchmarks[@]}"; do
        pattern="${benchmark}.cslab_4_2_branch_predictors.out"
        echo "Pattern: $pattern"
        # Iterate over files in the directory
        for file in "${Outputs}"/*; do
            # Check if the filename matches the pattern
            if [[ "$file" =~ $pattern ]]; then
                total_instructions=$(grep "Total Instructions:" "$file" | awk '{print $3}')
                # If it matches, take the information we need reading from the file
                incorrect_branches=$(grep "RAS ($entrie entries):" "$file" | awk '{print $5}')
                echo "we have incorrect = $incorrect_branches"
                # Calculate MPKI
                MPKI=$(echo "scale=6; (${incorrect_branches} * 1000) / ${total_instructions}" | bc)
                MPKI_sum=$(echo "scale=6; ${MPKI_sum} + ${MPKI}" | bc)
                # Increasing of files_count. At the end it should be 14 as we have 14 different benchmarks
                ((files_count++))
            fi
        done
    done
    mean_MPKI=$(echo "scale=6; ${MPKI_sum} / ${files_count}" | bc)
    # Store the results in the output file created
    echo "For RAS with ${entrie} entries and for all ${files_count} benchmarks we found that: " >> "$out_file"
    echo "Mean MPKI: ${mean_MPKI}" >> "$out_file"
done
```

Αναπαριστώντας τα αποτελέσματά μας σε ένα διάγραμμα παίρνουμε:



Παρατηρούμε ότι το MPKI καθώς το πλήθος των entries στην RAS αυξάνεται, φθίνει με μειούμενο ρυθμό. Δηλαδή, παρατηρούμε μία τεράστια μείωση στην αρχή, η οποία απ' το 16 και μετά γίνεται ολοένα και μικρότερη. Αν είχαμε σαν κριτήριο μόνο το MPKI, και άρα αποκλειστικά την επίδοση, θα επιλέγαμε σαν πλήθος εγγραφών τις 64. Ωστόσο, όπως ειπώθηκε και πριν, η αύξηση των εγγραφών αυξάνει και τις απαιτήσεις σε υλικό, αυξάνοντας δηλαδή και το κόστος. Συνεπώς θα ήταν συνετό να επιλέξουμε το πλήθος entries που αποφέρει μία πολύ καλή επίδοση με το ελάχιστο δυνατό κόστος. Αυτό το πλήθος είναι οι 16 εγγραφές στην RAS.