1η ΑΣΚΗΣΗ ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Ενρίκα Ηλιάνα Ματζόρι

AM:03120143

7.1.6 Ζητούμενα

Ζητούμενο 1.1

(7.1,7.1.2,7.1.3) Για κάθε μία απ' τις L1,L2,TLB ,για κάθε περίπτωση(configuration) και για θα κάθε benchmark, θα εκτελέσουμε το αντίστοιχο run.sh αρχείο(που παρατίθεται στη συνέχεια),που αποτελεί προσαρμογή του δοσμένου run_l1.sh αρχείου, ώστε να λάβουμε τις μετρήσεις που χρειαζόμαστε προκειμένου να εξάγουμε συμπεράσματα για την επίδοση της καθεμίας.

Προκειμένου να παρουσιάσουμε σε γραφική παράσταση τις μεταβολές των παραμέτρων σε κάθε περίπτωση, δεδομένου ότι έχουμε εκτελέσει το κάθε configuration για 8 διαφορετικά bechmarks, θα πρέπει για κάθε συνδυασμό παραμέτρων να συλλέξουμε τις πληροφορίες που χρειαζόμαστε και να πάρουμε έναν μέσο όρο IPC,MPKI λαμβάνοντας υπόψιν και τα 8 benchmarks σε κάθε περίπτωση.

Αξίζει να τονίσουμε ότι, για το IPC , εφόσον πρόκειται για Instructions/Cycle, πρόκειται για ένα μέγεθος που μετράει το πλήθος εμφάνισης ενός γεγονότος στη μονάδα του χρόνου (δηλαδή στον αριθμητή έχουμε ένα πλήθος από events-instructions και στον παρονομαστή έχουμε κύκλους-cycles,που αντιστοιχίζονται σε χρόνο). Συνεπώς, εφόσον πρόκειται για ένα μέγεθος-ρυθμό ,για να πάρουμε σαφή εικόνα για την επίδοση, θα υπολογίσουμε τον αρμονικό μέσο όρο. Στην περίπτωση του MPKI (Misses per 1000 Instructions) προχωράμε στον υπολογισμό του αριθμητικού μέσου όρου αφού δεν εμπλέκεται κάπου η έννοια του κύκλου/χρόνου αλλά αποτελεί καθαρό αριθμό(Misses/Instructions → events/events). Θεωρούμε τέλος ότι τα bechmarks έχουν πάντα την ίδια βαρύτητα, οπότε δεν απαιτείται η προσθήκη βαρών στους υπολογισμούς μας.

Συνεπώς, σε έναν ξεχωριστό φάκελο κάθε φορά βρίσκουμε τους μέσους όρους του IPC,MPKI λαμβάνοντας υπόψιν και τα 8 benchmarks για κάθε configuration ξεχωριστά.

Στην συνέχεια παρουσιάζουμε σε γραφική παράσταση τα αποτελέσματα μας.

Οι κώδικες για τις L1,L2,TLB είναι σχεδόν ίδιοι και το μόνο που αλλάζει κάθε φορά είναι οι σταθερές τιμές των παραμέτρων και τα configurations. Επομένως παρουσιάζουμε ενδεικτικά τα 3 αρχεία(run, mean calculation, plot) για την L1.

Open — run_l1.sh — ~/Downloads/advcomparch-ex1-helpcode — Save

```
#!/bin/bash

## Modify the following paths appropriately
PARSEC_PATH=/home/iliana/Downloads/parsec-3.0
PIN_EXE=/home/iliana/Downloads/pin-3.30-98830-g1d7b601b3-gcc-linux/pin
PIN_TOOL=/home/iliana/Downloads/advcomparch-ex1-helpcode/pintool/obj-intel64/simulator.so
CMDS_FILE=./cmds_simlarge.txt
outDir="./outputs/"

export LD_LIBRARY_PATH=$PARSEC_PATH/pkgs/libs/hooks/inst/amd64-linux.gcc-serial/lib/

## Triples of <cache_size>_<associativity>_<block_size>
CONFS="32_4_32 32_4_64 32_4_128 32_8_32 32_8_64 32_8_128 64_4_64 64_8_32 64_8_64 64_8_128 128_8_32
128_8_64 128_8_128"

L2size=2048
L2assoc=16
L2bsize=256
TLBe=64
TLBp=4096
TLBa=4
L2prf=0

#in order not to write everytime all benchmarks as arguments
benchmarks=("blackscholes" "bodytrack" "canneal" "fluidanimate" "freqmine" "rtview" "streamcluster"
"swaptions")

for BENCH in "${benchmarks[@]}"; do
        cmd=$(cat ${CMDS_FILE} | grep "$BENCH")
for conf in $CONFS; do
        ## Get parameters
    L1size=$(echo $conf | cut -d'_' -f1)
    L1assoc=$(echo $conf | cut -d'_' -f2)
    L1bsize=$(echo $conf | cut -d'_' -f3)

        outFile=$(printf "%s.dcache_cslab.L1_%04d_%02d_%03d.out" $BENCH ${L1size} ${L1assoc} $
{L1bsize})
        outFile="$outDir/$outFile"

        pin_cmd="$PIN_EXE -t $PIN_TOOL -o $outFile -L1c ${L1size} -L1a ${L1assoc} -L1b ${L1bsize} -
L2c ${L2size} -L2a ${L2assoc} -L2b ${L2bsize} -TLBe ${TLBe} -TLBp ${TLBp} -TLBa ${TLBa} -L2prf $
{L2prf} -- $cmd"
        time $pin_cmd
done
done
```

Το παρακάτω πρόγραμμα(και γενικά όλα τα mean.sh scripts) δημιουργεί συνολικά (#bencharks) * (#configuration) αρχεία της μορφής:

Open — L1_32_4_32_mean_... — ~/Downloads/parsec-3.0/... — Save

```
L1-Data Cache
Size(KB): 32
Block Size(B): 32
Associativity: 4
For this triplet in L1 and for all 8 files we found that:
Mean IPC: .450657
Mean MPKI: 15.375000
```

Open | mean_L1.sh ~/Downloads/advcomparch-ex1-helpcode | Save

```bash
#!/bin/bash

#  Script.sh
#
#
#  Created by Iliana on 13/4/24.
#
Outputs="/home/iliana/Downloads/parsec-3.0/parsec_workspace/outputs"
results="/home/iliana/Downloads/parsec-3.0/parsec_workspace/l1_mean_results"
# List of configurations
CONFS=(32_4_32 32_4_64 32_4_128 32_8_32 32_8_64 32_8_128 64_4_64 64_8_32 64_8_64 64_8_128 128_8_32 128_8_64 128_8_128)

# Iterate over each configuration
for conf in "${CONFS[@]}"; do

    # We set the Internal Field Separator to '_'(we need this to store variables using read)
    IFS='_'

    # Read the parameters into variables
    read size associativity bsize <<< "$conf"

    # Store the parameters into variables
    L1_cache_size="$size"
    L1_associativity="$associativity"
    L1_block_size="$bsize"

    # Print the stored variables
    echo "L1 cache size: $L1_cache_size"
    echo "L1 associativity: $L1_associativity"
    echo "L1 block size: $L1_block_size"

    # Initialization of the sums we will need to calculate the mean
    IPC_sum=0
    MPKI_sum=0
    files_count=0

    # Create the output file
    out_file="${results}/L1_${L1_cache_size}_${L1_associativity}_${L1_block_size}_mean_values.txt"

    echo "L1-Data Cache" >> "$out_file"

    # Write the parameters into the output file
    echo "Size(KB): $L1_cache_size" >> "$out_file"
    echo "Block Size(B): $L1_block_size" >> "$out_file"
    echo "Associativity: $L1_associativity" >> "$out_file"

    #pattern to match filenames
    pattern="L1_$(printf "%04d" "$L1_cache_size")_$(printf "%02d" "$L1_associativity")_$(printf "%03d" "$L1_block_size").out"
    echo "Pattern: $pattern"

    # Iterate over files in the directory
    for file in "${Outputs}"/*; do
        # Check if the filename matches the pattern
        if [[ "$file" =~ $pattern ]]; then
            # If it matches, take the information we need reading from the file
            total_instructions=$(grep "Total Instructions:" "$file" | awk '{print $3}')
            #total_cycles=$(grep "Total Cycles:" "$file" | awk '{print $3}')
            IPC=$(grep "IPC:" "$file" | awk '{print $2}')
            echo "IPC = $IPC"
            total_misses=$(grep "L1-Total-Misses:" "$file" | awk '{print $2}')

            # Calculate MPKI
            cM=$(echo "${total_misses} * 1000" | bc)
            MPKI=$(echo "${cM} / ${total_instructions}" | bc)

            # Add IPC and MPKI to the total sum
            ipc_inverted=$(bc -l <<< "scale=6; 1 / ${IPC}")


            echo "for iteration $files_count IPC_sum= $IPC_sum and 1/IPC = $ipc_inverted "
            IPC_sum=$(bc <<< "${IPC_sum} + ${ipc_inverted}")
            MPKI_sum=$(bc <<< "${MPKI_sum} + ${MPKI}")

            # Increasing of files_count. At the end it should be 8 as we have 8 different benchmarks
            ((files_count++))
        fi
    done

    # For each triplet(13 in total) we calculate mean IPC and mean MPKI for the 8 benchmarks
    mean_IPC=$(bc <<< "scale=6; ${files_count} / ${IPC_sum}")
    mean_MPKI=$(bc <<< "scale=6; ${MPKI_sum} / ${files_count}")

    # Store the results in the output file created
    echo "For this triplet in L1 and for all ${files_count} files we found that: " >> "$out_file"
    echo "Mean IPC: ${mean_IPC}" >> "$out_file"
    echo "Mean MPKI: ${mean_MPKI}" >> "$out_file"
done
```

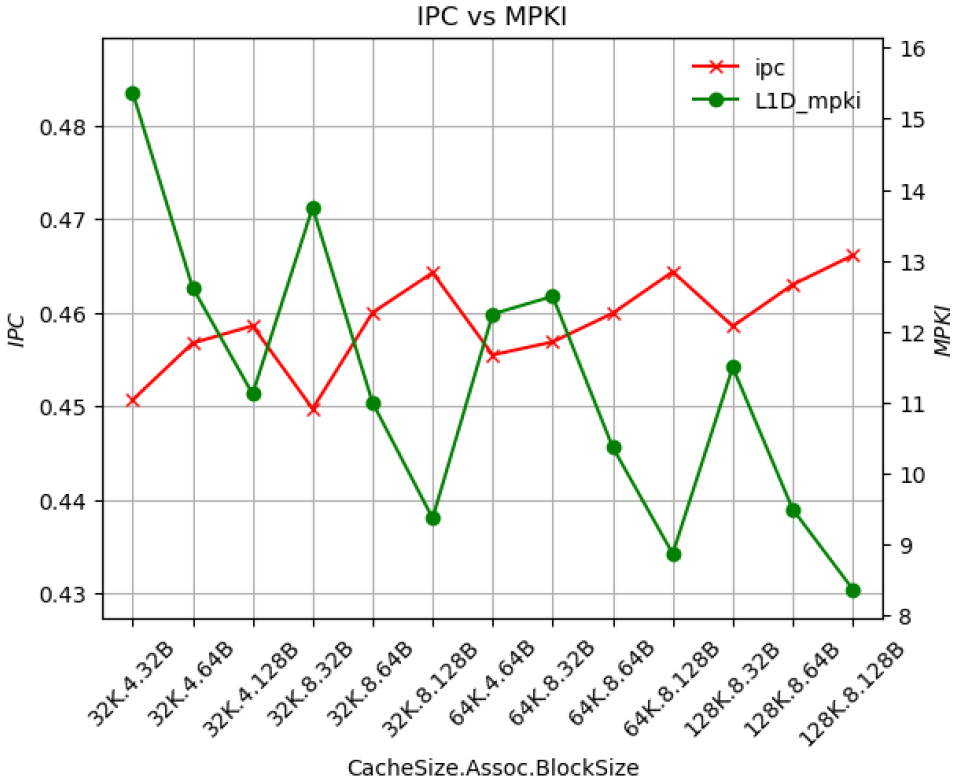
Open

plot_l1_final.sh
~/Downloads/advcomparch-ex1-helpcode

Save

```
#!/usr/bin/env python3

import sys
import numpy as np

## We need matplotlib:
## $ apt-get install python-matplotlib
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

x_Axis = []
ipc_Axis = []
mpki_Axis = []

for outFile in sys.argv[1:]:
        fp = open(outFile)
        line = fp.readline()
        while line:
                tokens = line.split()
                if (line.startswith("Mean IPC:")):
                        ipc = float(tokens[2])
                elif (line.startswith("L1-Data Cache")):
                        sizeLine = fp.readline()
                        l1_size = sizeLine.split()[1]
                        bsizeLine = fp.readline()
                        l1_bsize = bsizeLine.split()[2]
                        assocLine = fp.readline()
                        l1_assoc = assocLine.split()[1]
                elif (line.startswith("Mean MPKI:")):
                        mpki = float(tokens[2])


                line = fp.readline()

        fp.close()

        l2ConfigStr = '{}K.{}.{}B'.format(l1_size,l1_assoc,l1_bsize)
        print (l2ConfigStr)
        x_Axis.append(l2ConfigStr)
        ipc_Axis.append(ipc)
        mpki_Axis.append(mpki)

print (x_Axis)
print (ipc_Axis)
print (mpki_Axis)

fig, ax1 = plt.subplots()
ax1.grid(True)
ax1.set_xlabel("CacheSize.Assoc.BlockSize")

xAx = np.arange(len(x_Axis))
ax1.xaxis.set_ticks(np.arange(0, len(x_Axis), 1))
ax1.set_xticklabels(x_Axis, rotation=45)
ax1.set_xlim(-0.5, len(x_Axis) - 0.5)
ax1.set_ylim(min(ipc_Axis) - 0.05 * min(ipc_Axis), max(ipc_Axis) + 0.05 * max(ipc_Axis))
ax1.set_ylabel("$IPC$")
line1 = ax1.plot(ipc_Axis, label="ipc", color="red",marker='x')

ax2 = ax1.twinx()
ax2.xaxis.set_ticks(np.arange(0, len(x_Axis), 1))
ax2.set_xticklabels(x_Axis, rotation=45)
ax2.set_xlim(-0.5, len(x_Axis) - 0.5)
ax2.set_ylim(min(mpki_Axis) - 0.05 * min(mpki_Axis), max(mpki_Axis) + 0.05 *
  max(mpki_Axis))
ax2.set_ylabel("$MPKI$")
line2 = ax2.plot(mpki_Axis, label="L1D_mpki", color="green",marker='o')

lns = line1 + line2
labs = [l.get_label() for l in lns]

plt.title("IPC vs MPKI")
lgd = plt.legend(lns, labs)
lgd.draw_frame(False)
plt.savefig("L1_mean_final.png",bbox_inches="tight")
```

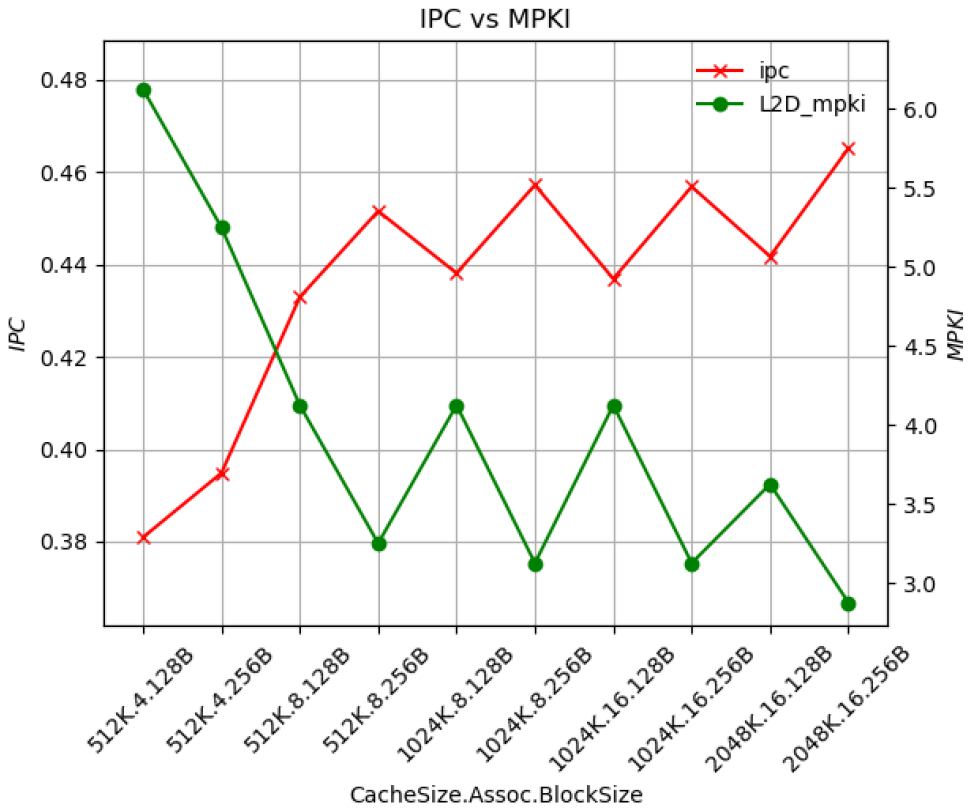
- Οπότε τελικά για την **L1** έχουμε:



Όπως φαίνεται από την γραφική απεικόνιση, ο συνδυασμός παραμέτρων που μας δίνει καλύτερη επίδοση για την L1,παρόλο που δεν είναι πολύ καλύτερος από άλλους κοντινούς του, δηλαδή με αυτόν πετυχαίνουμε την μεγαλύτερη τιμή του IPC(κατά μέσο όρο για όλα τα benchmarks) είναι ο εξής:
L1 size(KB) = 128
L1 associativity = 4
L1 cache block size(B) = 64

Βάσει της γραφικής απεικόνισης έχουμε και κάποιες συμπληρωματικές παρατηρήσεις σχετικά με την σχέση των μεγεθών στην επίδοση της cache. Συγκεκριμένα, λαμβάνοντας υπόψιν τα 3 πρώτα configurations, όπου το μέγεθος της cache και η συσχετικότητα είναι σταθερά, όταν το **μέγεθος του block** αυξάνεται, τότε αυξάνεται και το IPC, οπότε και η επίδοση. Αυτό είναι αναμενόμενο, λόγω της χωρικής τοπικότητας των δεδομένων(spatial locality), δηλαδή της αρχής που μας λέει ότι γειτονικά στοιχεία όσων έχουν ήδη προσπελαστεί, έχουν αυξημένη πιθανότητα να προσπελαστούν στο άμεσο μέλλον. Συνεπώς, ένα μεγαλύτερου μεγέθους μπλοκ της cache συμβάλλει ακριβώς σε αυτό-φέρνουμε περισσότερα γειτονικά δεδομένα- μειώνοντας έτσι τις αστοχίες. Ένα υπερβολικά μεγάλο μέγεθος block ωστόσο ίσως δεν είναι τόσο αποδοτικό λόγω του μεγάλου όγκου δεδομένων προς αναζήτηση, αυτό όμως με τις δικές μας τιμές δεν φαίνεται.
Συγκρίνοντας τα αποτελέσματα που λαμβάνουμε όταν το μόνο μέγεθος που μεταβάλλεται είναι αυτό της **συσχετικότητας** μπορούμε εύκολα να εξάγουμε το συμπέρασμα ότι όταν αυτή αυξάνεται, η επίδοση είναι καλύτερη. Αυτό μοιάζει απόλυτα λογικό αφού, σε αντίθεση με την κρυφή μνήμη άμεσης απεικόνισης όπου ένα μπλοκ μπορεί να τοποθετηθεί σε μία και μόνο συγκεκριμένη θέση, η συσχετιστική κρυφή μνήμη συνόλου(set-associative cache) προσφέρει μία πιο ευέλικτη τοποθέτηση των μπλοκ, καθώς κάθε μπλοκ μπορεί να τοποθετηθεί σε οποιαδήποτε θέση του συνόλου που του αντιστοιχεί. Συνεπώς ο αριθμός των αστοχιών είναι σαφώς μικρότερος και άρα επιτυγχάνεται καλύτερη επίδοση.
Τέλος, όταν ο μόνος παράγοντας που μεταβάλλεται είναι το **μέγεθος της cache** αναμενόμενα βλέπουμε ότι το IPC αυξάνεται και αντιστοίχως το MPKI μειώνεται καθώς η πιθανότητα hit σε μία μεγαλύτερη cache(αφού μπορούμε να έχουμε περισσότερα δεδομένα σε αυτήν) είναι μεγαλύτερη.

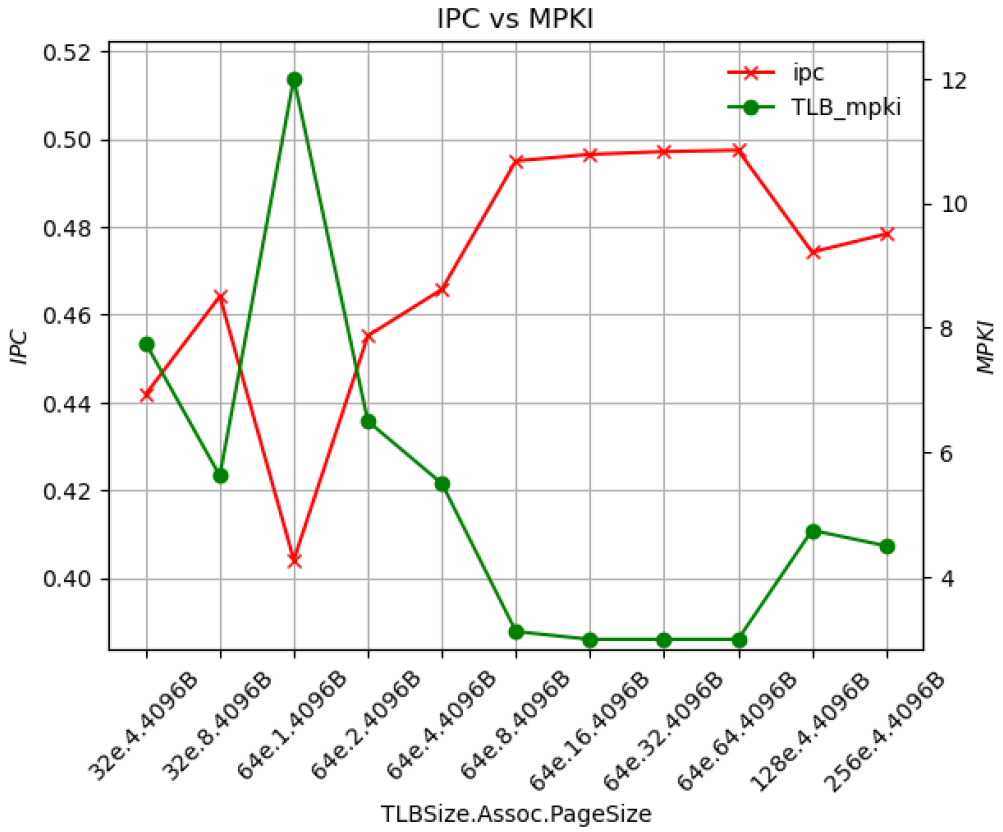
- Για την **L2**, χρησιμοποιώντας τον συνδυασμό που δίνει την καλύτερη L1 cache παίρνουμε:



Είναι προφανές ότι την καλύτερη L2 την έχουμε για τον εξής συνδυσαμό παραμέτρων:
L2 size(KB) = 2048
L2 associativity = 16
L2 cache block size(B) = 256

Τα συμπεράσματα που εξάγαμε για τον ρόλο του κάθε μεγέθους για την L1 ισχύει και σε αυτήν την περίπτωση. Επισημαίνεται ότι, ενώ η αύξηση της συσχετικότητας στις τριάδες {512.4.256},{512,8,256} αποφέρει μία πολύ μεγάλη αύξηση στην τιμή του IPC,και άρα της επίδοσης, στην συνέχεια δεν βλέπουμε κάποια εξίσου μεγάλη άνοδο της τιμής του λόγω της αύξησης της πολυπλοκότητας της cache.

- Για το **TLB**(κρυφή μνήμη αναζήτησης μετάφρασης) παίρνουμε τις επιλεχθείσες-βέλτιστες τιμές των L1 και L2 οπότε έχουμε:



Απ' τα configurations που διαφέρουν μεταξύ τους μόνο στο **associativity** μπορούμε να εξάγουμε το συμπέρασμα ότι όταν αυτό αυξάνεται, το IPC, άρα και η επίδοση, επίσης αυξάνεται. Αυτό είναι αναμενόμενο και με βάση όσα ειπώθηκαν παραπάνω. Ωστόσο, παρατηρούμε ότι για τα configurations {64.8.4096}, {64.16.4096}, {64.32.4096}, {64.64.4096} ο ρυθμός αύξησης του IPC μειώνεται συγκριτικά με το πώς αυξανόταν όταν οι τιμές τις συσχετικότητας ήταν μικρότερες(για ίδιες τιμές size, block size).Αυτό ίσως οφείλεται στο ότι σε μεγάλες τιμές συσχετικότητας αυξάνεται η πολυπλοκότητα της cache(και άρα οι απαιτήσεις από θέμα hardware είναι μεγαλύτερες) καθώς και η πολυπλοκότητα των προσπελάσεων, θυσιάζοντας έτσι χρόνο σε αυτές.

Σε ό,τι αφορά την αύξηση του **TLB size**, απ'το διάγραμμα πορκύπτει ότι όταν το μέγεθος αυξάνεται, το IPC επίσης αυξάνεται καθώς έτσι αυξάνονται και οι πιθανότητες hit. Ωστόσο πάλι ο ρυθμός αύξησης IPC μειώνεται όσο αυξάνεται το μέγεθος της TLB και, όπως περιγράφηκε παραπάνω, αυτό συμβαίνει λόγω της αυξημένης πολυπλοκότητας και του μεγαλύτερου χρόνου προσπέλασης.

Για το TLB page size δεν μπορούμε να εξάγουμε κάποιο συμπέρασμα αφού σε όλα τα configurations η παράμετρος αυτή είναι ίδια.

Από την γραφική προκύπτει ότι την καλύτερη επίδοση(υψηλότερο IPC) την έχουμε για τον εξής συνδυασμό παραμέτρων του TLB:
TLB size(entries) = 64
TLB associativity = 64
TLB page size(B) = 4096

(7.1.4) Μία τακτική μείωσης των misses και ,ως εκ τούτου, αύξηση της επίδοσης είναι η προανάκληση**(prefetching).** Αυτό που συνέβαινε έως τώρα με την παράμετρο L2prf ίση με 0 ήταν ότι όταν γινόταν ένα L1 miss, ελέγχαμε την L2 και αν και εκεί συνέβαινε ένα miss τότε φέρναμε στην cache το block που αναζητούσαμε. Αυτό που θέλουμε να κάνουμε τώρα είναι κάθε φορά που φτάνουμε να έχουμε μία αστοχία στην L2, να φέρνουμε στην cache όχι μόνο το block που προκάλεσε την αστοχία, αλλά και τα N επόμενά του, για διαφορετικές τιμές του N(στον κώδικα αυτές είναι το _l2_prefetched_lines).

Συνεπώς για να το κάνουμε αυτό μεταβάλλουμε τον κώδικα που αφορά το prefetching στο αρχείο cache.h ως εξής:

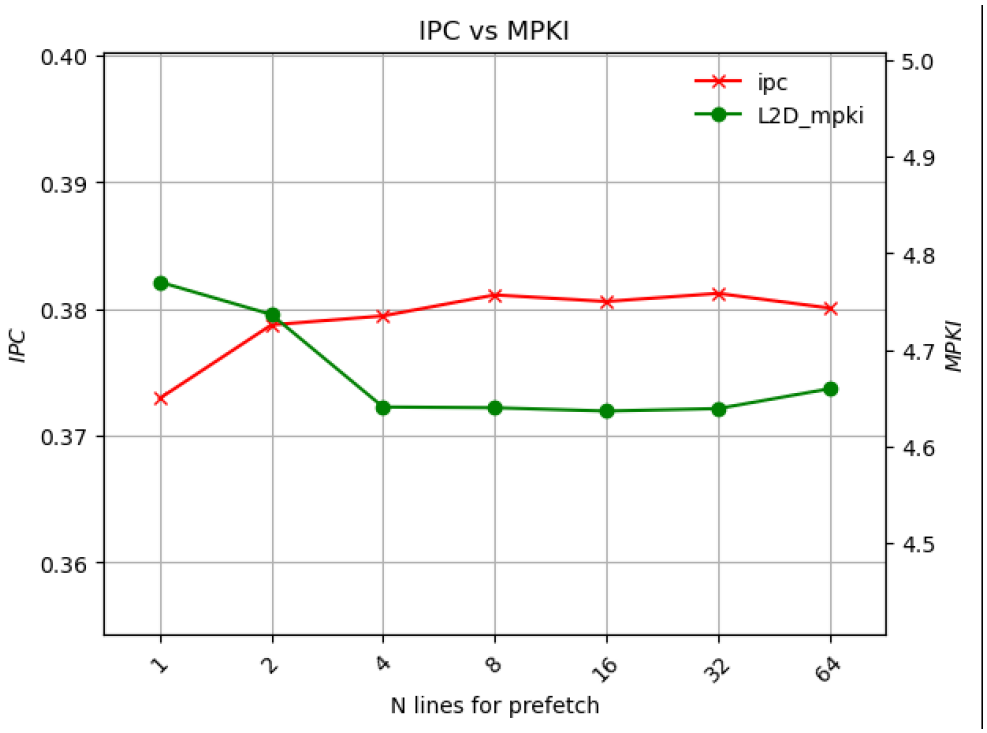
```
// PREFETCHING
ADDRINT prefetch_addr = addr;
//before we bring a block into L2 we have to check that this is not yet in L2
for (UINT32 i=0; i < _l2_prefetch_lines; i++) {
    prefetch_addr += L2BlockSize();
    CACHE_TAG l2prefTag;
    UINT32 l2prefSetIndex;
    // Let's check L2
    SplitAddress(prefetch_addr, L2LineShift(),L2SetIndexMask(), l2prefTag, l2prefSetIndex);
    SET & l2prefSet = _l2_sets[l2prefSetIndex];
    bool l2prefHit = l2prefSet.Find(l2prefTag);
    // If the address is not in L2 cache, prefetch it
    if (!l2prefHit) { //we have a miss so the block is not in L2 and we bring it
            CACHE_TAG l2Prefetch_replaced = l2prefSet.Replace(l2prefTag);
            // If L2 is inclusive and a TAG has been replaced we need to remove
            // all evicted blocks from L1.
            if ((L2_INCLUSIVE == 1) && !(l2Prefetch_replaced == INVALID_TAG)) {
                    ADDRINT replacedAddr = ADDRINT(l2Prefetch_replaced) << FloorLog2(L2NumSets());
                    replacedAddr = replacedAddr | l2SetIndex;
                    replacedAddr = replacedAddr << L2LineShift();
                    for (UINT32 i=0; i < L2BlockSize(); i+=L1BlockSize()) {
                            ADDRINT newAddr = replacedAddr | i;
                            SplitAddress(newAddr, L1LineShift(), L1SetIndexMask(), l1Tag, l1SetIndex);
                            l1Set = _l1_sets[l1SetIndex];
                            l1Set.DeleteIfPresent(l1Tag);
                    }
            }

    }
```

Πρακτικά στον κώδικα αν έχουμε ένα L2 miss τότε φέρνουμε το block που μας το προκάλεσε. Αν η L2 είναι γεμάτη και δεν υπάρχει ελεύθερος χώρος τότε ελευθερώνουμε χώρο σύμφωνα με την πρακτική αντικατάστασης που ορίζει το κάθε σύστημα, εδώ αυτή είναι η LRU. Για τα επόμενα blocks τσεκάρουμε αν βρίσκονται ήδη στην L2(ώστε να μην φέρουμε κάτι που ήδη υπάρχει) και φέρνουμε όποιο block (από τα N) δεν υπάρχει ήδη. Στην συνέχεια ελέγχουμε αν η L2 είναι inclusive, αν δεν είναι το prefetching του συγκεκριμένου block ολοκληρώθηκε. Σε αντίθετη περίπτωση ελέγχουμε ότι το block που διαγράφηκε(εάν διαγράψαμε κάποιο block για την απελευθέρωση χώρου στην L2) δεν υπάρχει στην L1, και αν υπάρχει να το διαγράψουμε και από εκεί.

Συνεπώς, με τις βέλτιστες τιμές των L1,L2 και τις ενδεικνούμενες τιμές για το TLB, εκτελούμε το πρόγραμμα run_pref.sh που παρατίθεται παρακάτω :

```
#!/bin/bash

## Modify the following paths appropriately
PARSEC_PATH=/home/iliana/Downloads/parsec-3.0
PIN_EXE=/home/iliana/Downloads/pin-3.30-98830-g1d7b601b3-gcc-linux/pin
PIN_TOOL=/home/iliana/Downloads/advcomparch-ex1-helpcode/pintool/obj-intel64/simulator.so
CMDS_FILE=./cmds_simlarge.txt
outDir="./pref_outputs/"

export LD_LIBRARY_PATH=$PARSEC_PATH/pkgs/libs/hooks/inst/amd64-linux.gcc-serial/lib/

## Triples of <prefetched lines>
N="1 2 4 8 16 32 64"

L1size=128
L1assoc=8
L1bsize=128
L2size=2048
L2assoc=16
L2bsize=256
TLBe=64
TLBp=4096
TLBa=4
#L2prf=0

#in order not to write everytime all benchmarks as arguments
benchmarks=("blackscholes" "canneal" "fluidanimate" "rtview")

for BENCH in "${benchmarks[@]}"; do
        cmd=$(cat ${CMDS_FILE} | grep "$BENCH")
for n in $N; do
        ## Get parameters
    L2prf=$n
    #L1size=$(echo $conf | cut -d'_' -f1)
    #L1assoc=$(echo $conf | cut -d'_' -f2)
    #L1bsize=$(echo $conf | cut -d'_' -f3)

        outFile=$(printf "%s.dcache_cslab.L2prf_%02d.out" $BENCH ${L2prf})
        outFile="$outDir/$outFile"

        pin_cmd="$PIN_EXE -t $PIN_TOOL -o $outFile -L1c ${L1size} -L1a ${L1assoc} -L1b ${L1bsize} -L2c ${L2size} -L2a ${L2assoc} -L2b ${L2bsize} -TLBe ${TLBe} -TLBp ${TLBp} -TLBa ${TLBa} -L2prf ${L2prf} -- $cmd"
        time $pin_cmd
done
done
```

Και οπότε υπολογίζοντας κανονικά τους μέσους όρους για τα 4 benchmarks όπως κάναμε στα προηγούμενα ερωτήματα παίρνουμε την παρακάτω γραφική παράσταση για την απόδοση της L2 για τις διάφορες τιμές του N:



Όπως και ήταν αναμενόμενο, καθώς το πλήθος των blocks που προανακλώνται στην cache μεγαλώνει, το IPC αυξάνεται, αφού αυξήσαμε έτσι τις πιθανότητες για hit στην L2 εκμεταλλευόμενοι την χωρική τοπικότητα που αναφέραμε παραπάνω. Ωστόσο βλέπουμε ότι η απόδοση πέφτει όταν ο αριθμός των blocks που φέρνουμε εκ των προτέρων αυξάνεται κατά πολύ. Αυτό πιθανά να οφείλεται στις καθυστερήσεις που υπάρχουν για την εισροή των blocks στην cache, οι οποίες πλέον είναι τόσο μεγάλες που ξεπερνούν το κέρδος σε χρόνο που έχουμε από τη μείωση των αστοχιών.

Είναι σημαντικό να παρατηρήσουμε ότι το IPC που πετυχαίνουμε εδώ στη βέλτιστη περίπτωση (για N= IPC=) είναι μικρότερο απ' το IPC= 0.465114 που είχαμε προηγουμένως χωρίς την τακτική του prefetching για τις ίδιες παραμέτρους για την L2. Ωστόσο πρέπει να θυμόμαστε ότι στα στοιχεία που λαμβάνουμε για την απόδοση της L2 με prefetching παίρνουμε υπόψιν μόνο 4 benchmarks και όχι 8 όπως προηγουμένως. Μάλιστα, αν κοιτάξουμε αναλυτικά τα αποτελέσματα για τα 4 benchmarks με βάση τα οποία εξετάζουμε την πρακτική του prefetching θα βλέπαμε πολύ διαφορετικά αποτελέσματα στα IPC τους. Για παράδειγμα για N
Συνεπώς **για τα benchmarks μπορούμε να συμπεράνουμε** ότι το καθένα διαχειρίζεται και εκμεταλλεύεται διαφορετικά την χωρική τοπικότητα(spatial locality) με αποτέλεσμα να θέλει διαφορετική διαχείριση για καλά αποτελέσματα. Συνεπώς τα benchmarks είναι όλα ισοδύναμα όπως ειπώθηκε και στην αρχή, αλλά σε καμία περίπτωση δεν είναι ίσα. Αυτό ήταν δυσκολότερο να το συμπεράνουμε στις μετρήσεις για τα L1,L2,TLB καθώς οι τιμές που έδιναν τα benchmarks ήταν πολύ παρόμοιες και οδηγούσαν σε πιο «ξεκάθαρα» συμπεράσματα.

(7.1.5) Ένα πολύ σημαντικό ζήτημα στην ιεραρχία της μνήμης είναι αυτό που πραγματεύεται τον χειρισμό των αστοχιών και πιο συγκεκριμένα, τις διάφορες πολιτικές αντικατάστασης ενός μπλοκ στην κρυφή μνήμη όταν δεν υπάρχει χώρος για κάποιο νέο μπλοκ. Μέχρι τώρα στις μετρήσεις μας, η μέθοδος που χρησιμοποιούνταν ήταν η LRU (least recently used) όπου το μπλοκ που αντικαθίσταται είναι αυτό που μένει αχρησιμοποίητο για το μεγαλύτερο χρονικό διάστημα. Η αντικατάσταση LRU υλοποιείται με την παρακολούθηση του πότε χρησιμοποιήθηκε κάθε στοιχείο του συνόλου σε σχέση με τα υπόλοιπα στοιχεία του. Εδώ αυτή τη συγκεκριμένη λειτουργία έχει η συνάρτηση find ενώ η όλη διαδικασία με την συνάρτηση Replace μέσα στην κλάση LRU.

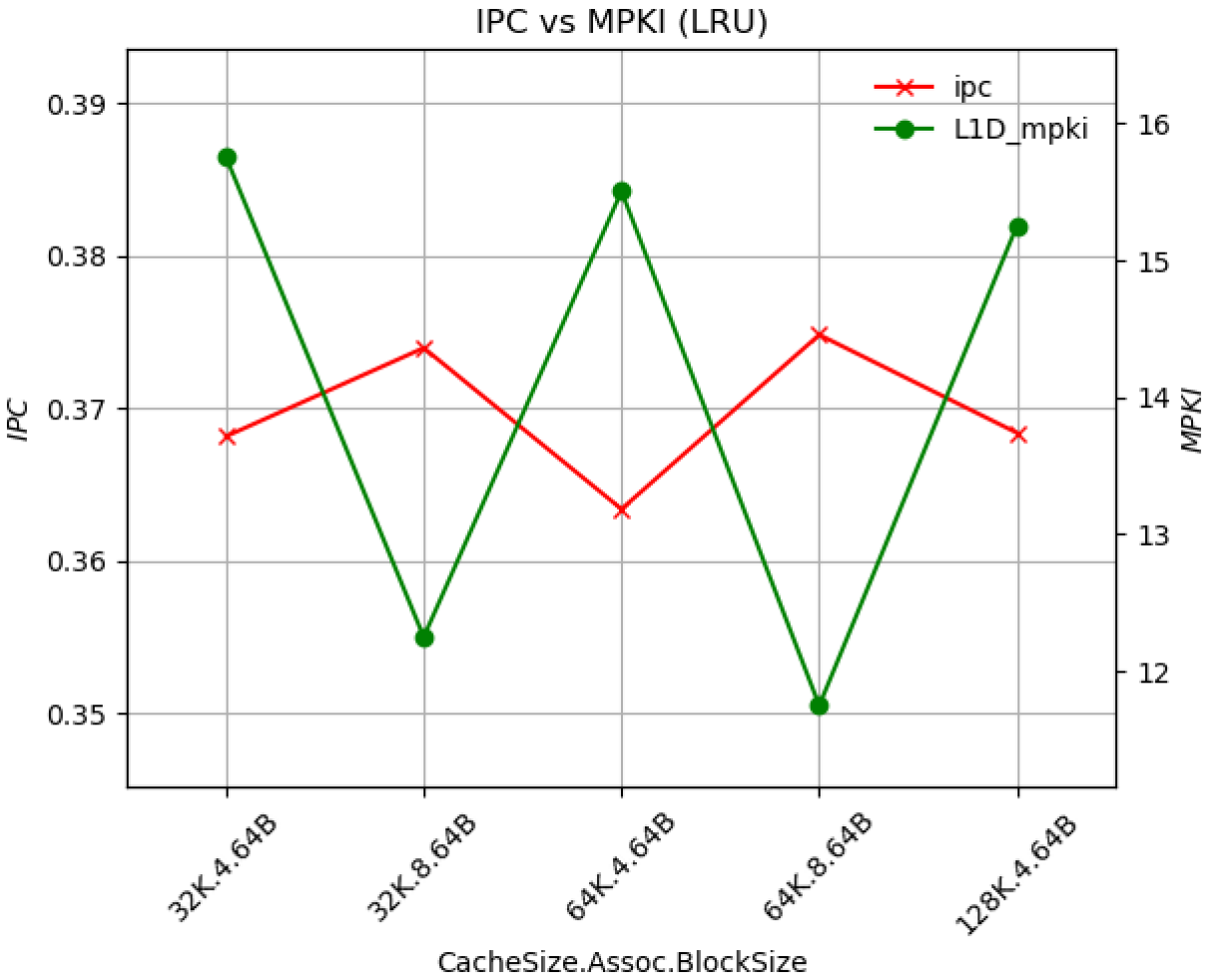
Καλούμαστε λοιπόν εδώ να τροποποιήσουμε αυτές τις δύο συναρτήσεις ώστε το μπλοκ που θα αντικαθίσταται κάθε φορά να είναι τυχαίο. Συνεπώς οι δύο συναρτήσεις παίρνουν την παρακάτω μορφή:

```
UINT32 Find(CACHE_TAG tag)
{
    for (std::vector<CACHE_TAG>::iterator it = _tags.begin();
         it != _tags.end(); ++it)
    {
    //in random replace policy we don't need to keep track of most/less recent used blocks
        if (*it == tag) { // Tag found, lets make it MRU

            //_tags.erase(it);
            //_tags.push_back(tag);
            return true;
        }
    }

    return false;
}
```

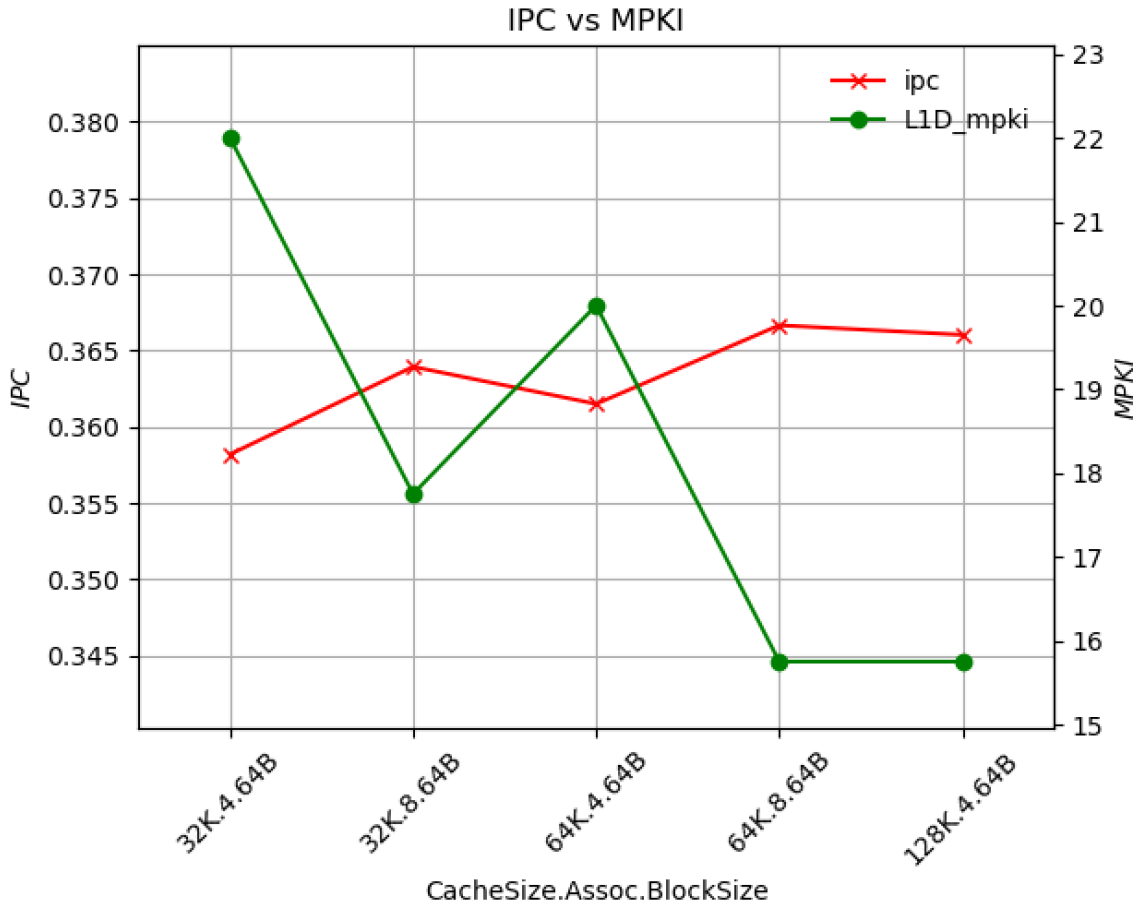
```
//implementation of random replacement policy
    CACHE_TAG Replace(CACHE_TAG tag)
    {
        CACHE_TAG ret = INVALID_TAG;
        _tags.push_back(tag);
        if (_tags.size() > _associativity) {
            srand(20143); //AM:03120143
            int random_number = rand();
            int random_index = random_number % _tags.size();
            //we have to erase now the randomly chosen block
            //so we remove the tag at the randomly chosen index from the _tags vector
            ret = _tags[random_index];
            _tags.erase(_tags.begin() + random_index);
        }
        return ret;
    }
```

Επομένως για την βέλτιστη τιμή της L2 και τις δοθείσες παραμέτρους της TLB, για διαφορετικά configurations για την L1 έχουμε τα εξής αποτελέσματα:

Όταν χρησιμοποιούμε πολιτική αντικατάστασης **LRU**:



Όταν χρησιμοποιούμε **Random Replacement** policy:



Στην συγκεκριμένη περίπτωση, για όρισμα στην srand ίσο με 20143, βλέπουμε ότι γενικά η LRU δουλεύει καλύτερα(με ορισμένες εξαιρέσεις όπου έχουμε πολύ λιγότερα misses στην τυχαία επιλογή), καθώς πετυχαίνουμε μεγαλύτερα IPC's αλλά χωρίς μεγάλη διαφορά από την Τυχαία Επιλογή. Γενικά βλέπουμε ότι, και στις δύο περιπτώσεις, η απόδοση είναι αυξημένη όταν η συσχετικότητα είναι μεγάλη. Αυτό ακούγεται λογικό αφού η μεγαλύτερη συσχετικότητα συνεπάγεται και μικρότερη χρήση των πολιτικών αντικατάστασης, αφού το κάθε block θα έχει περισσότερες επιλογές θέσεων στις οποίες μπορεί να τοποθετηθεί στην cache.

Ζητούμενο 1.2

Με βάση τα παραπάνω που παίρναμε τις τιμές για τις L1,L2 μόνο με βάση την απόδοση, και άρα το μεγαλύτερο IPC, τότε οι επιλεγμένες τιμές για τις δυο κρυφές μνήμες είναι οι εξής:

L1:
Size(KB) = 128
Associativity = 4
Block Size(B) = 128

L2:
Size(KB) = 2048
Associativity = 16
Block Size(B) = 256

Ωστόσο, για την σχεδίαση του συστήματος δεν φτάνει μόνο να λαμβάνουμε υπόψιν την απόδοση, αλλά και το κόστος κατασκευής και την κατανάλωση ηλεκτρικής ισχύος. Αυτά τείνουν να μεγαλώνουν όταν αυξάνουμε την χωρητικότητα και την πολυπλοκότητα της cache. Συνεπώς καλούμαστε να σκεφτούμε κάποιους συνδυασμούς παραμέτρων, που να έχουν κοντινή απόδοση με τον βέλτιστο, αλλά χωρίς να είναι τόσο κοστοβόρα σε χωρητικότητα και πολυπλοκότητα. Παρατηρώντας και τις γραφικές, βλέπουμε ότι υπάρχουν τιμές που είναι πολύ κοντινές στις ιδανικές(οι οποίες όπως προείπαμε δεν

απέχουν και πολύ απ' τις υπόλοιπες στις περισσότερες περιπτώσεις), που δίνονται όμως και από πιο συνδυασμούς που δεν απαιτούν ακριβή αρχιτεκτονική.
Συνεπώς επιλέγουμε τα εξής:

L1:
Size(KB) = 128
Associativity = 8
Block Size(B) = 32

L2:
Size(KB) = 512
Associativity = 8
Block Size(B) = 256

Ζητούμενο 2

Στα προηγούμενα ερωτήματα που θεωρούσαμε τον κύκλο σταθερό είδαμε ότι όταν διπλασιαζόταν κάποια από τις παραμέτρους, τις περισσότερες φορές -αν όχι όλες- η απόδοση αυξανόταν. Δηλαδή βλέπαμε αύξηση του IPC η οποία συνοδευόταν από μείωση του MPKI.

Ωστόσο, εφόσον πλέον όταν αυξάνεται κάποια παράμετρος υπεισέρχονται καθυστερήσεις και δεδομένου ότι το IPC δίνεται απ' τον τύπο $IPC = \frac{\#instructions}{\#cycles}$, οι καθυστερήσεις αυτές προφανώς δημιουργούνε μείωση της απόδοσης, αναιρώντας έτσι ως έναν βαθμό, μικρό ή μεγάλο, την θετική επίδραση στην απόδοση που θα είχε η αύξηση κάποιας εκ των παραμέτρων.

Επομένως, πλέον για την επιλογή των παραμέτρων πρέπει να λάβουμε υπόψιν μερικά συγκεκριμένα σημεία:

- Αν διπλασιαστεί η χωρητικότητα της cache και η συσχετικότητα θα έχουμε καθυστερήσεις.
- Ο διπλασιασμός της χωρητικότητας επιφέρει πιο μεγάλες καθυστερήσεις.
- Ο διπλασιασμός του μέγεθος του block δεν επιφέρει καθυστερήσεις, άρα μπορούμε να το εκμεταλλευτούμε αυτό χωρίς όμως να είναι πολύ μεγάλο δίοτι
- Γενικά πρέπει να αποφευχθούν τα μεγάλα μεγέθη διότι προκαλούν καθυστερήσεις.

Με βάση λοιπόν τα συμπεράσματα απ' τις μετρήσεις που λάβαμε στα ποηγούμενα ερωτήματα διαισθητικά επιλέγουμε τις εξής τιμές για τις 2 cache:

L1:
Size(KB) = 32
Associativity = 8
Block Size(B) = 128

L2:
Size(KB) = 512
Associativity = 8
Block Size(B) = 256