

Application 1:

This application takes a number and puts it through a random function to calculate a floating-point value. The magic function, $\text{magicFunc}(n)$, is equal to $\text{magicFunc}(n - 1) - \text{magicFunc}(n - 2)/4$. The initial value is set in the data section on the code. This first program will use a smaller cache size. The initial value is set to 15.

Code:

```
# ECE365 Project Phase 3 Application 1

# Alex Santana

.data
num: .word 15      # index

.text
main:

    lw      $a0, num
    jal     magicFunc

    li      $v0, 2      # print float

    mtc1    $zero, $f6    # clear
    cvt.s.w $f6, $f6      # convert

    add.s   $f12, $f6, $f30 # print number return
    syscall

#END PROGRAM----

    li      $v0, 10
    syscall

#-----
```

magicFunc:

```
    beq    $a0, $zero, mfReturn0
    slti   $t0, $a0, 2
    bne    $t0, $zero, mfReturn1
    j      mfCalc
```

mfReturn0:

```
    add    $t0, $zero, $zero
    j      mfReturn
```

mfReturn1:

```
    addi   $t0, $zero, 1
    j      mfReturn
```

mfReturn:

```
    mtc1   $t0, $f30
    cvt.s.w $f30, $f30
    jr     $ra
```

mfCalc:

```
    mtc1   $zero, $f6      # clear float
    cvt.s.w $f6, $f6      # convert

    addi   $sp, $sp, -72
    sw     $ra, 0($sp)
    sw     $a0, 4($sp)
```

swc1 \$f0, 8(\$sp)

swc1 \$f2, 40(\$sp)

addi \$a0, \$a0, -1

jal magicFunc

add.s \$f0, \$f6, \$f30

lw \$a0, 4(\$sp)

addi \$a0, \$a0, -2

jal magicFunc

add.s \$f2, \$f6, \$f30

addi \$t0, \$zero, 4

mtc1 \$t0, \$f4

cvt.s.w \$f4, \$f4 # convert

div.s \$f2, \$f2, \$f4

sub.s \$f0, \$f0, \$f2

add.s \$f30, \$f6, \$f0

lw \$ra, 0(\$sp)

lw \$a0, 4(\$sp)

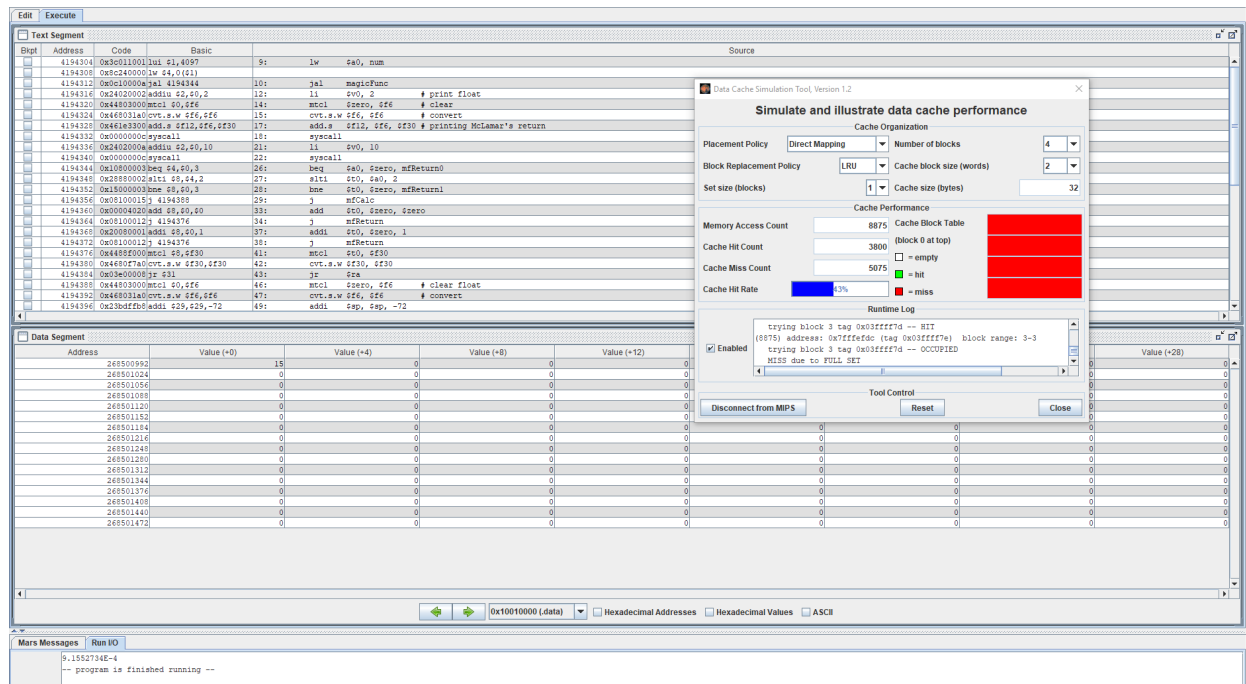
lwc1 \$f0, 8(\$sp)

lwc1 \$f2, 40(\$sp)

addi \$sp, \$sp, 72

jr \$ra

Direct Mapping



Number of blocks: 4

Cache block size: 2

Memory access count: 8875

Cache hit count: 3800

Cache miss count: 5075

Cache hit ratio: 43%

Fully Associative

The screenshot displays the Data Cache Simulation Tool interface. The main window shows a MIPS assembly program with instructions and their corresponding addresses. A pop-up window titled "Simulate and illustrate data cache performance" provides a summary of the simulation results.

Cache Organization:

- Placement Policy: Fully Associative
- Number of blocks: 4
- Block Replacement Policy: LRU
- Cache block size (words): 2
- Set size (Blocks): 4
- Cache size (Bytes): 32

Cache Performance:

- Memory Access Count: 8875
- Cache Hit Count: 4841
- Cache Miss Count: 4034
- Cache Hit Rate: 55%

Cache Block Table:

Block	Tag	Value	State
0	0x00000000	0	empty
1	0x00000000	0	empty
2	0x00000000	0	empty
3	0x00000000	0	empty

Data Segment:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
26f501092	18	0	0	0	0	0	0	0
26f501094	0	0	0	0	0	0	0	0
26f501096	0	0	0	0	0	0	0	0
26f501098	0	0	0	0	0	0	0	0
26f50109a	0	0	0	0	0	0	0	0
26f50109c	0	0	0	0	0	0	0	0
26f50109e	0	0	0	0	0	0	0	0
26f5010a0	0	0	0	0	0	0	0	0
26f5010a2	0	0	0	0	0	0	0	0
26f5010a4	0	0	0	0	0	0	0	0
26f5010a6	0	0	0	0	0	0	0	0
26f5010a8	0	0	0	0	0	0	0	0
26f5010aa	0	0	0	0	0	0	0	0
26f5010ac	0	0	0	0	0	0	0	0
26f5010ae	0	0	0	0	0	0	0	0
26f5010b0	0	0	0	0	0	0	0	0
26f5010b2	0	0	0	0	0	0	0	0
26f5010b4	0	0	0	0	0	0	0	0
26f5010b6	0	0	0	0	0	0	0	0
26f5010b8	0	0	0	0	0	0	0	0
26f5010ba	0	0	0	0	0	0	0	0
26f5010bc	0	0	0	0	0	0	0	0
26f5010be	0	0	0	0	0	0	0	0
26f5010c0	0	0	0	0	0	0	0	0
26f5010c2	0	0	0	0	0	0	0	0
26f5010c4	0	0	0	0	0	0	0	0
26f5010c6	0	0	0	0	0	0	0	0
26f5010c8	0	0	0	0	0	0	0	0
26f5010ca	0	0	0	0	0	0	0	0
26f5010cc	0	0	0	0	0	0	0	0
26f5010ce	0	0	0	0	0	0	0	0
26f5010d0	0	0	0	0	0	0	0	0
26f5010d2	0	0	0	0	0	0	0	0
26f5010d4	0	0	0	0	0	0	0	0
26f5010d6	0	0	0	0	0	0	0	0
26f5010d8	0	0	0	0	0	0	0	0
26f5010da	0	0	0	0	0	0	0	0
26f5010dc	0	0	0	0	0	0	0	0
26f5010de	0	0	0	0	0	0	0	0
26f5010e0	0	0	0	0	0	0	0	0
26f5010e2	0	0	0	0	0	0	0	0
26f5010e4	0	0	0	0	0	0	0	0
26f5010e6	0	0	0	0	0	0	0	0
26f5010e8	0	0	0	0	0	0	0	0
26f5010ea	0	0	0	0	0	0	0	0
26f5010ec	0	0	0	0	0	0	0	0
26f5010ee	0	0	0	0	0	0	0	0
26f5010f0	0	0	0	0	0	0	0	0
26f5010f2	0	0	0	0	0	0	0	0
26f5010f4	0	0	0	0	0	0	0	0
26f5010f6	0	0	0	0	0	0	0	0
26f5010f8	0	0	0	0	0	0	0	0
26f5010fa	0	0	0	0	0	0	0	0
26f5010fc	0	0	0	0	0	0	0	0
26f5010fe	0	0	0	0	0	0	0	0
26f501100	0	0	0	0	0	0	0	0
26f501102	0	0	0	0	0	0	0	0
26f501104	0	0	0	0	0	0	0	0
26f501106	0	0	0	0	0	0	0	0
26f501108	0	0	0	0	0	0	0	0
26f50110a	0	0	0	0	0	0	0	0
26f50110c	0	0	0	0	0	0	0	0
26f50110e	0	0	0	0	0	0	0	0
26f501110	0	0	0	0	0	0	0	0
26f501112	0	0	0	0	0	0	0	0
26f501114	0	0	0	0	0	0	0	0
26f501116	0	0	0	0	0	0	0	0
26f501118	0	0	0	0	0	0	0	0
26f50111a	0	0	0	0	0	0	0	0
26f50111c	0	0	0	0	0	0	0	0
26f50111e	0	0	0	0	0	0	0	0
26f501120	0	0	0	0	0	0	0	0
26f501122	0	0	0	0	0	0	0	0
26f501124	0	0	0	0	0	0	0	0
26f501126	0	0	0	0	0	0	0	0
26f501128	0	0	0	0	0	0	0	0
26f50112a	0	0	0	0	0	0	0	0
26f50112c	0	0	0	0	0	0	0	0
26f50112e	0	0	0	0	0	0	0	0
26f501130	0	0	0	0	0	0	0	0
26f501132	0	0	0	0	0	0	0	0
26f501134	0	0	0	0	0	0	0	0
26f501136	0	0	0	0	0	0	0	0
26f501138	0	0	0	0	0	0	0	0
26f50113a	0	0	0	0	0	0	0	0
26f50113c	0	0	0	0	0	0	0	0
26f50113e	0	0	0	0	0	0	0	0
26f501140	0	0	0	0	0	0	0	0
26f501142	0	0	0	0	0	0	0	0

The bottom of the window shows the "Mars Messages" panel with the following output:

```

p.1552734E-4
-- program is finished running --
  
```

Number of blocks: 4

Cache block size: 2

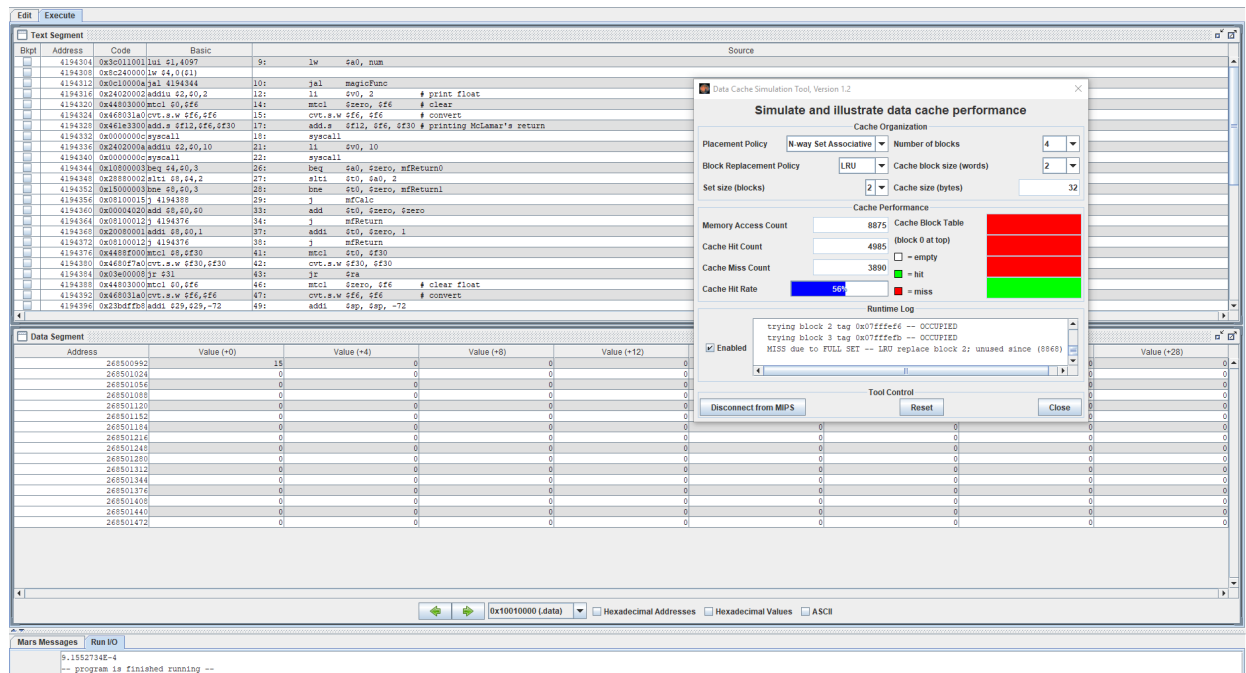
Memory access count: 8875

Cache hit count: 4841

Cache miss count: 4034

Cache hit ratio: 55%

Set Associative



Number of blocks: 4

Cache block size: 2

Memory access count: 8875

Cache hit count: 4985

Cache miss count: 3890

Cache hit ratio: 56%

Conclusion

This program help visualize how mapping affects performance. In this program the number 15 was put into a function. This allowed the program to return the same value after stepping through the same steps through all 3 mapping techniques. Because the function only did arithmetic, I set the blocks and size to a smaller amount so that it may use the cache more rigorously. This helped depict the different in hit ration among the set techniques and direct. Direct was obviously returned the lowest hit ratio. The difference between fully and set differed by only a percent as the instructions were not as complex. Even with these restrictions, the performance increase may still be seen amount the different techniques.

Application 2:

This application accesses an array and returns information such as steps size and rep count in order to better visualize cache operation and performance. Because of the random nature of the program, hit ratio was not constant but stayed in a range among different mappings.

Code:

```
# ECE365 Project Phase 3 Application 2

# Alex Santana

.data
arr:    .space 2048          # max array size in BYTES

.text
main:
    li    $a0, 256          # array size in BYTES
    li    $a1, 8             # step size
    li    $a2, 2             # rep count
    li    $a3, 1

    jal    wordAccess

    #END PROGRAM----

    li    $v0, 10
    syscall

    #-----

wordAccess:
    la    $s0, arr          # array pointer
    addu   $s1, $s0, $a0      # array limit
```

```
sll    $t1, $a1, 2        # inc step
```

wordLP:

```
move   $s6, $a0
move   $s7, $a1
move   $s5, $v0
addiu  $a0, $0, 100       # seed for random number generator
addiu  $a1, $a1, 0
addiu  $v0, $0, 42        # syscall 42 is random int range
syscall
sll    $a0, $a0, 2        # offset
addu   $s4, $s0, $a0      # move pointer
sw     $0, 0($s4)         # array[(index+offset)/4] = 0
move   $a0, $s6
move   $a1, $s7
move   $v0, $s5
```

wordCheck:

```
addu   $s0, $s0, $t1      # increment ptr
blt    $s0, $s1, wordLP
addi   $a2, $a2, -1
bgtz   $a2, wordAccess
jr     $ra
```

byteAccess:

```
la     $s0, arr           # array pointer
addu   $s1, $s0, $a0      # array limit
```

byteLP:


```
beq    $a3, $0, byteZero
```

```
lbu     $t0, 0($s0)          # inc index
```

```
addi    $t0, $t0, 1
```

```
sb      $t0, 0($s0)
```

```
j       byteCheck
```

byteZero:

```
sb      $0, 0($s0)          # reset index
```

byteCheck:

```
addu    $s0, $s0, $a1        # inc pointer
```

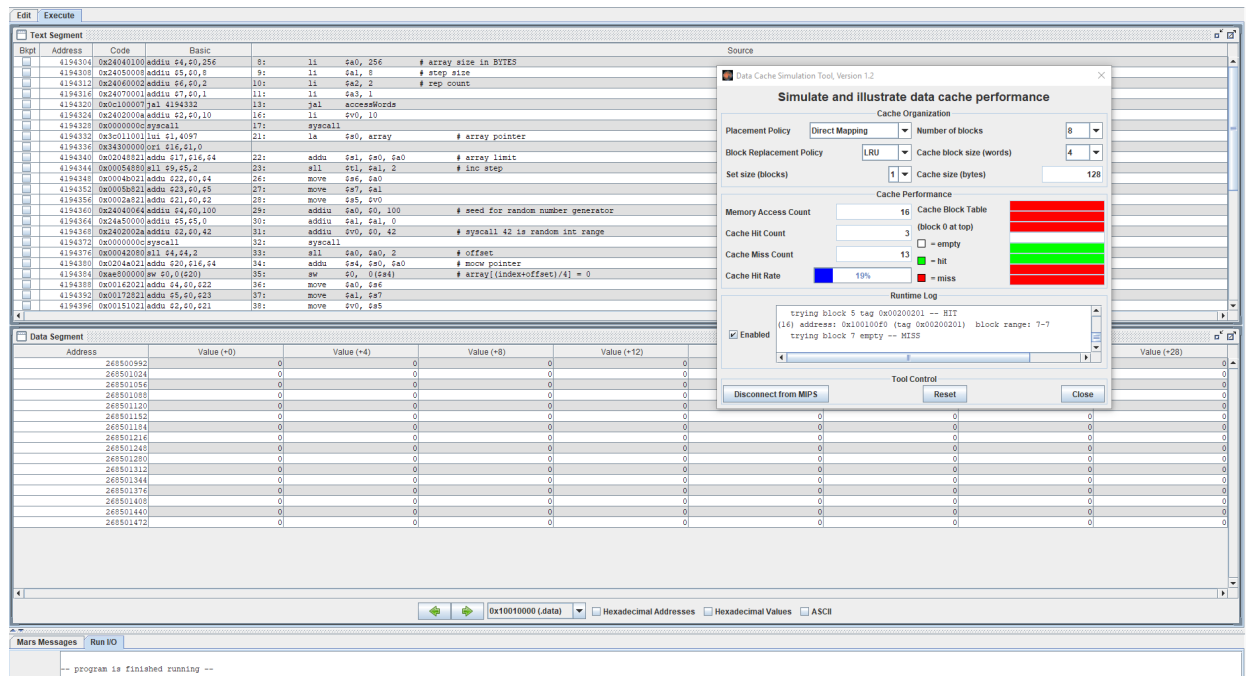
```
blt     $s0, $s1, byteLP
```

```
addi    $a2, $a2, -1
```

```
bgtz    $a2, byteAccess
```

```
jr      $ra
```

Direct Mapping



Number of blocks: 8

Cache block size: 4

Memory access count: 16

Cache hit count: 3

Cache miss count: 13

Cache hit ratio: 19%

Fully Associative

The screenshot displays the Data Cache Simulation Tool interface. The main window shows MIPS assembly code for a program that simulates a cache. The code includes instructions for setting up the cache, accessing data, and calculating the hit rate. A simulation window is open, showing the following parameters:

- Cache Organization:** Fully Associative, Number of blocks: 8, Cache block size (words): 4, Cache size (bytes): 128.
- Block Replacement Policy:** LRU.
- Cache Performance:** Memory Access Count: 16, Cache Hit Count: 5, Cache Miss Count: 11, Cache Hit Rate: 31%.
- Runtime Log:**
 - trying block 5 tag 0x0100100e -- OCCUPIED
 - trying block 6 tag 0x0100100e -- OCCUPIED
 - trying block 7 tag 0x0100100e -- HIT

The Data Segment window shows the following data values:

Address	Value (-8)	Value (-4)	Value (-8)	Value (-12)	Value (-28)
26f500992	0	0	0	0	0
26f501024	0	0	0	0	0
26f501056	0	0	0	0	0
26f501088	0	0	0	0	0
26f501120	0	0	0	0	0
26f501152	0	0	0	0	0
26f501184	0	0	0	0	0
26f501216	0	0	0	0	0
26f501248	0	0	0	0	0
26f501280	0	0	0	0	0
26f501312	0	0	0	0	0
26f501344	0	0	0	0	0
26f501376	0	0	0	0	0
26f501408	0	0	0	0	0
26f501440	0	0	0	0	0
26f501472	0	0	0	0	0

Number of blocks: 8

Cache block size: 4

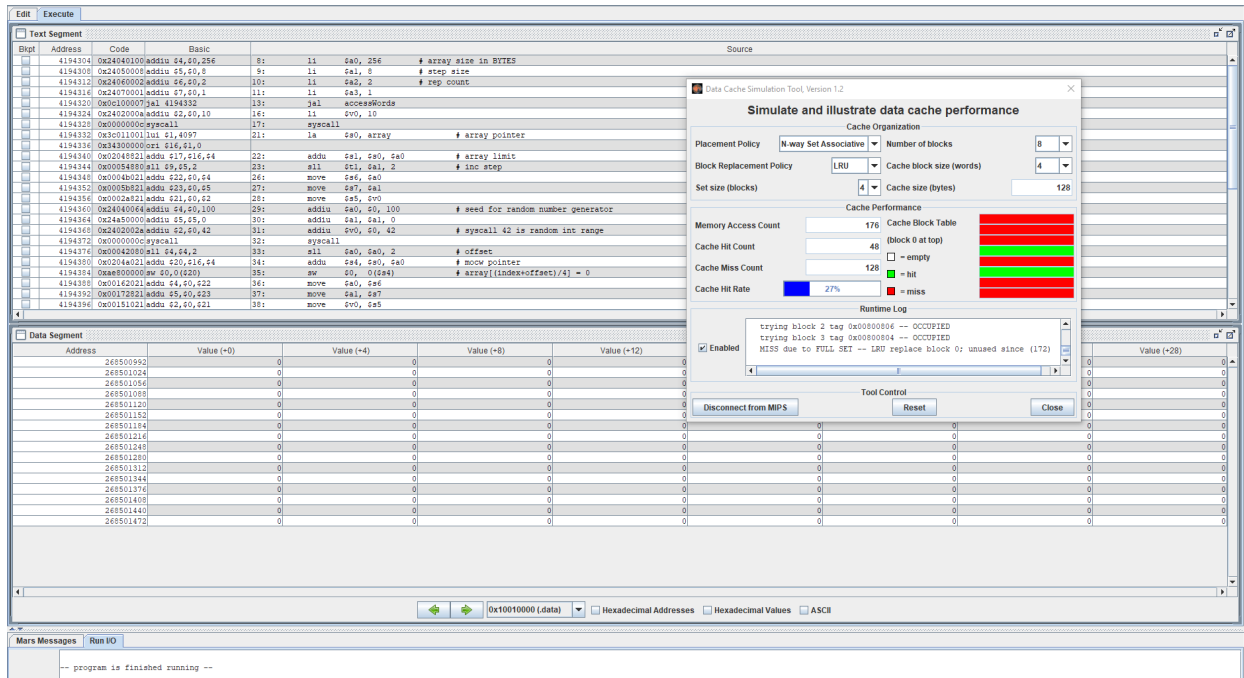
Memory access count: 16

Cache hit count: 5

Cache miss count: 11

Cache hit ratio: 31%

Set Associative



Number of blocks: 8

Cache block size: 4

Memory access count: 176

Cache hit count: 48

Cache miss count: 128

Cache hit ratio: 27%

Conclusion

I noticed the hit rate was non-deterministic. I believe this was due to the random nature of the program, but I was able to still conclude that improvement was attained when moving away from direct mapping. Because of the small number of blocks and size, direct mapping resulted in a hit ratio of under 20% as the rewriting of the block in the small case cause misses repeatedly. Upon changing mapping, I noticed an increase in hit count and ratio as I was testing different number of sets. Fully associative and set associative mappings gave similar results but not as close as the previous program.