

The background features a series of concentric circles in light gray, some solid and some dashed, creating a ripple effect. A large, solid purple oval is centered on the page, containing the title and author information. A thick, dark gray curved line sweeps across the bottom left, partially overlapping the purple oval.

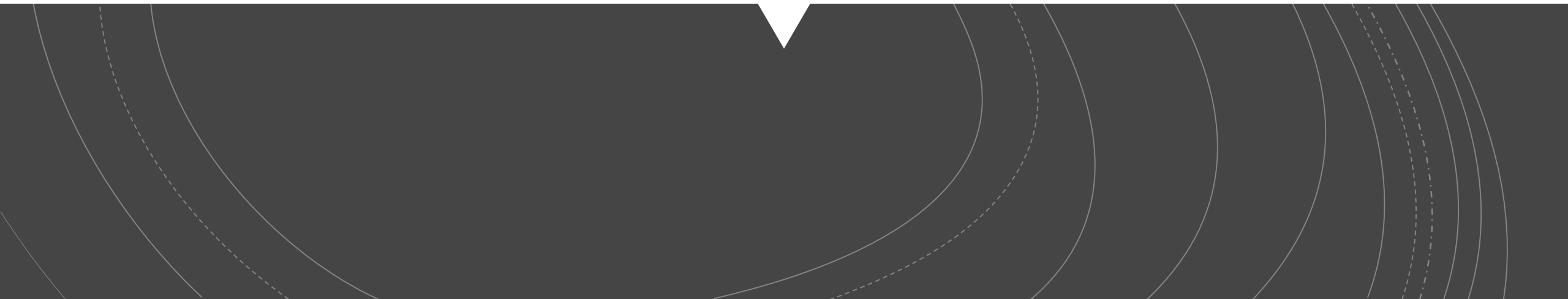
# MARS: Caching Evaluation

Cristina McLaughlin

# Overview

1. Introduction
2. Cache Overview
3. Testing Locality with MARS Simulator
4. Fibonacci Two Ways
5. Conclusion

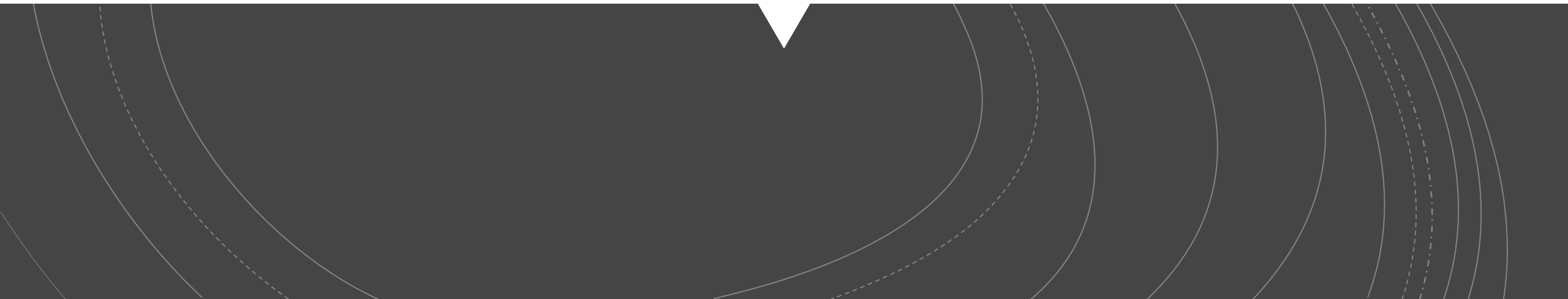
# 1. Introduction



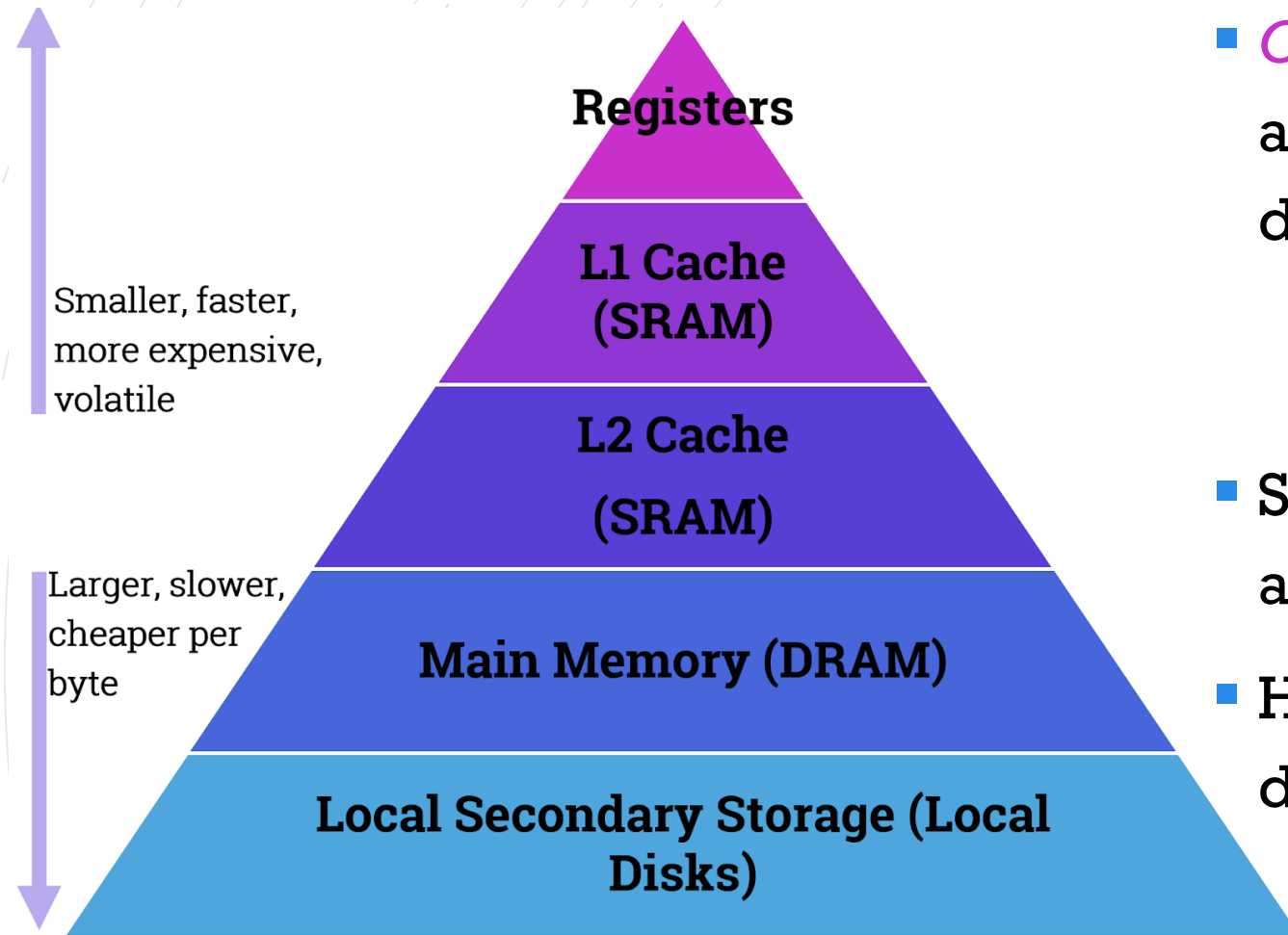
# Introduction

- Goals
  - Understand cache memory organization/operation
  - Understand performance of caches
  - Evaluate Fibonacci algorithm
    - Iteration vs Recursion

## 2. Cache Overview



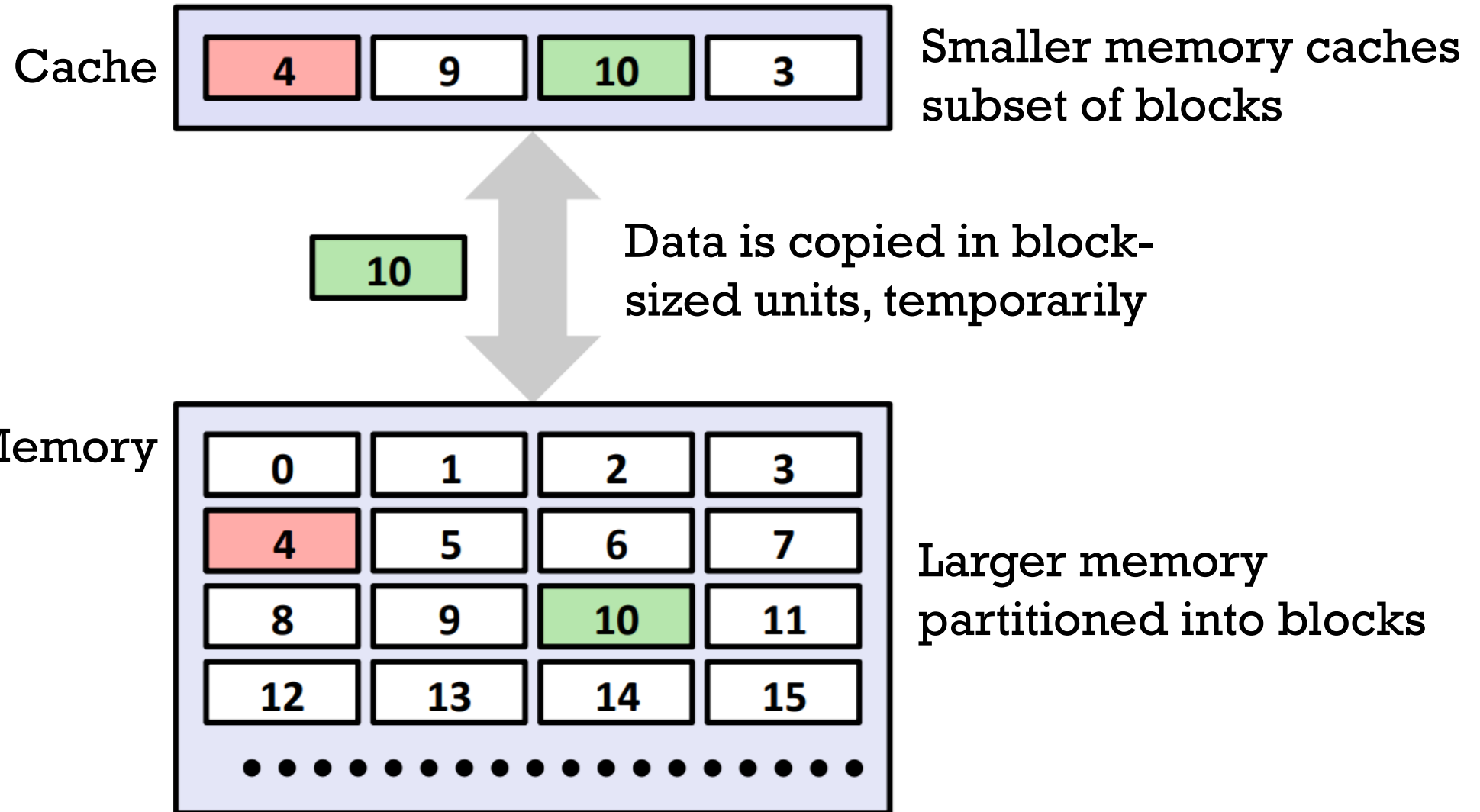
# Caches and Memory Hierarchy



- **Cache**: smaller, faster storage device to act as data staging area for larger, slower device
  - memory contents stored closer to point of use
- Small SRAM-based memory managed automatically by hardware
- Holds frequently requested data/instructions
  - immediately available when needed

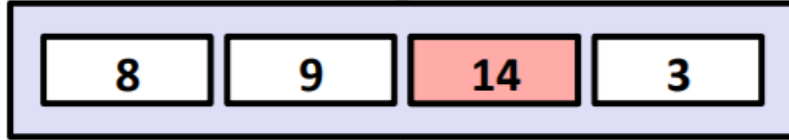
# General Cache Concepts

- When the processor needs to read/write a location in memory it first checks for the entry in the cache



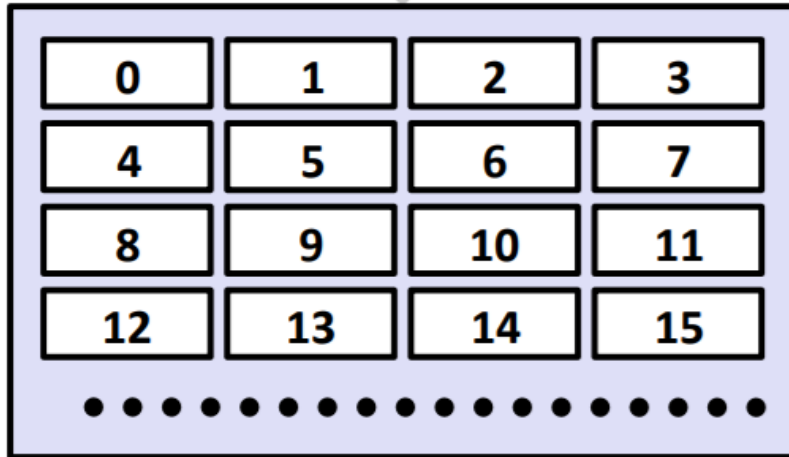
Processor requests 14

Cache



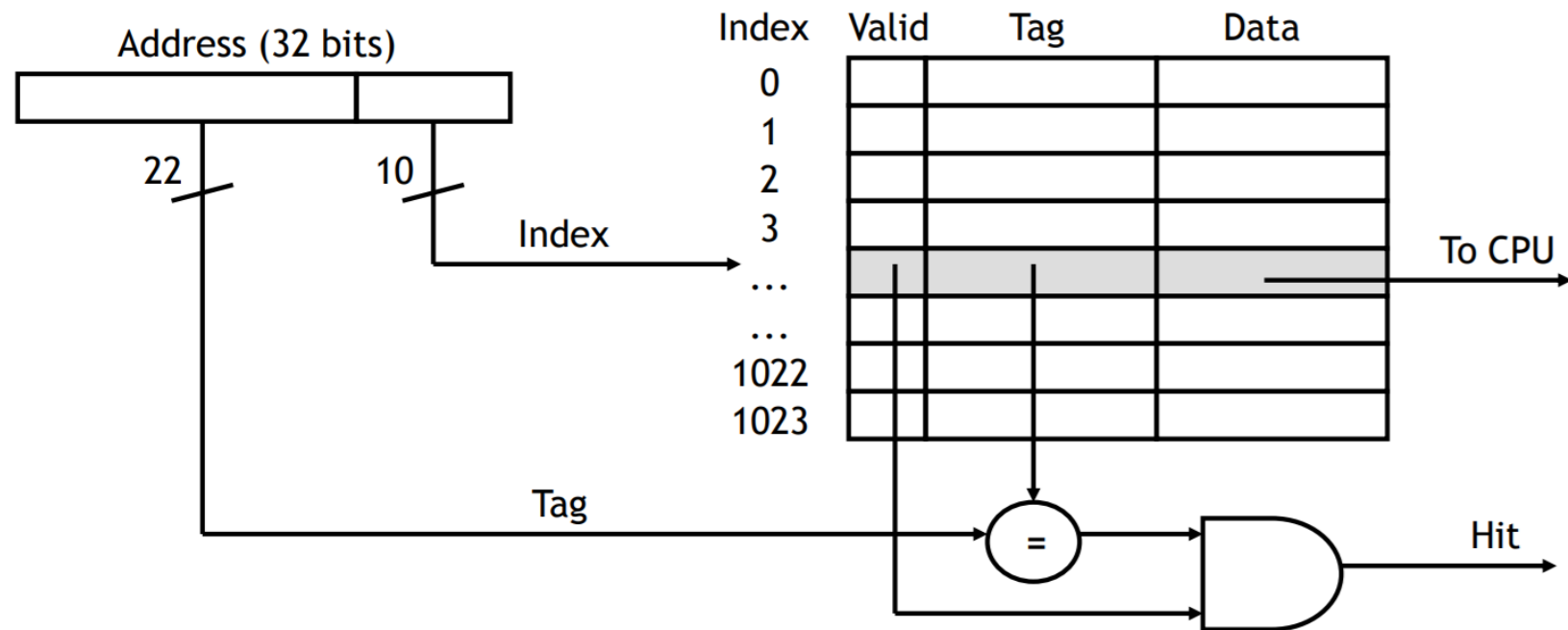
Data in block is  
needed  
14 is in cache:  
**HIT**

Memory



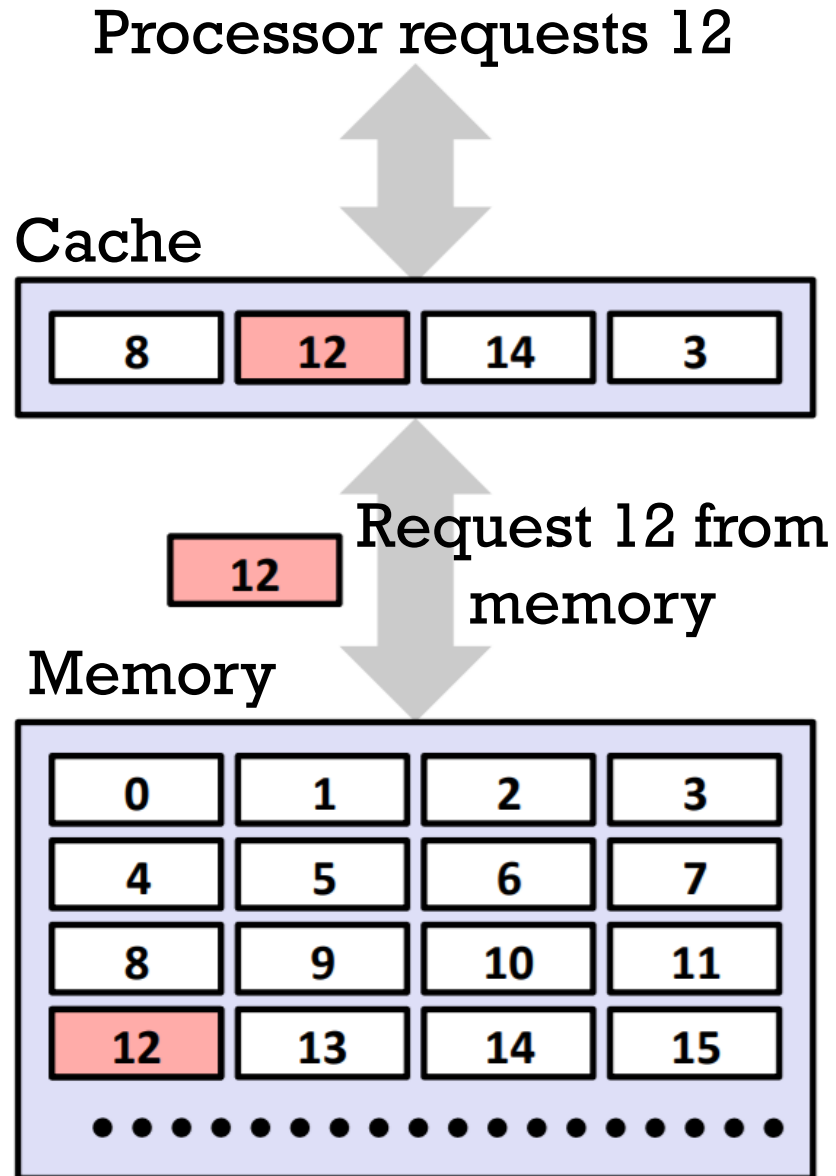
## Cache Hit

- Fast access from cache
- If block is valid  $\wedge$  tag matches upper bits of address = HIT
- Data will be sent to CPU





# Cache Miss



Data in block is needed  
12 is not in cache: **MISS**

Block is fetched from memory

Block is stored back into cache

- *Placement*: determines where block goes in cache
- *Replacement*: determines which block is evicted

# Principle of Locality

- *Principle of Locality*: Programs tend to use data/instructions with addresses near or equal to recently used ones
  - *Temporal Locality (Time)*
    - Recently referenced blocks are likely to be called again in the near future
  - *Spatial Locality (Space)*
    - Items within a neighborhood of addresses tend to be referenced close together in time

## Temporal Locality

- Data
  - Reference the variable **sum** each iteration
- Instruction
  - Cycle through loop repeatedly

## Spatial Locality

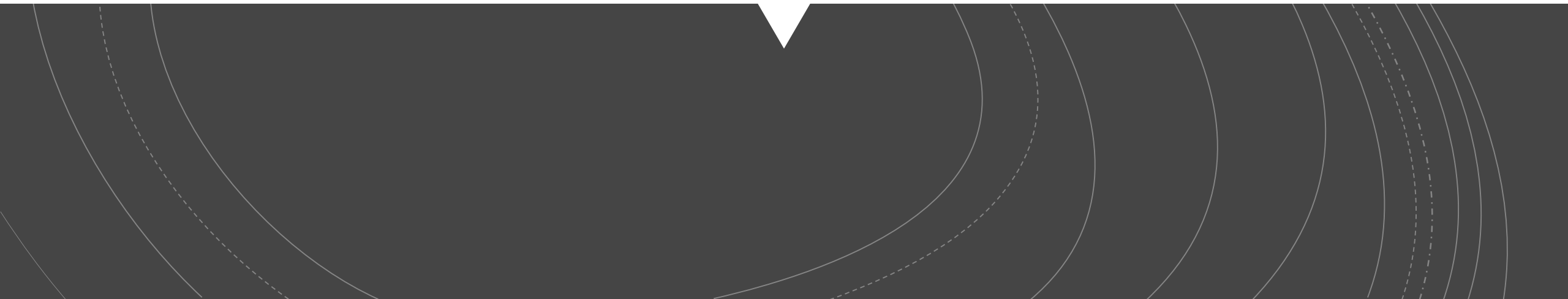
- Data
  - Array elements are accessed in succession
- Instruction
  - Loop means instructions are referenced in same sequence

*Cache-Friendly Code*: programs that try to keep access close together to minimize cache misses, through optimizing aspects of locality

## Locality Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

# 3. Testing Locality with MARS Simulator



- Matrices are a great way to see caching at work
- Algorithm to traverse through a 16x16 matrix assigned values in order
  - 256 different elements = 256 memory accesses
  - Row traversal vs. Column Traversal
- *Algorithms Analysis Perspective:*
  - No difference
  - $O(n^2)$  time
- *Cache Perspective:*
  - Very different
  - Row traversal uses principles of locality

## Row Major Traversal

```
for (row = 0; row < 16; row++)  
    for (col = 0; col < 16; col++)  
        data[row][col] = num++;
```

## Column Major Traversal

```
for (col = 0; col < 16; col++)  
    for (row = 0; row < 16; row++)  
        data[row][col] = num++;
```

# MARS Cache Simulator – Row Major Traversal

**Simulate and illustrate data cache performance**

**Cache Organization**

Placement Policy:  Number of blocks:

Block Replacement Policy:  Cache block size (words):

Set size (blocks):  Cache size (bytes):

**Cache Performance**

Memory Access Count:  Cache Block Table: 

(block 0 at top)

Cache Hit Count:  ☐ = empty

Cache Miss Count:  ☒ = hit

Cache Hit Rate:  ☒ = miss

**Runtime Log**

☒ Enabled

**Visualizing memory reference patterns**

Show unit boundaries (grid marks) ☒ **Spatial Locality**

Memory Words per Unit:

Unit Width in Pixels:

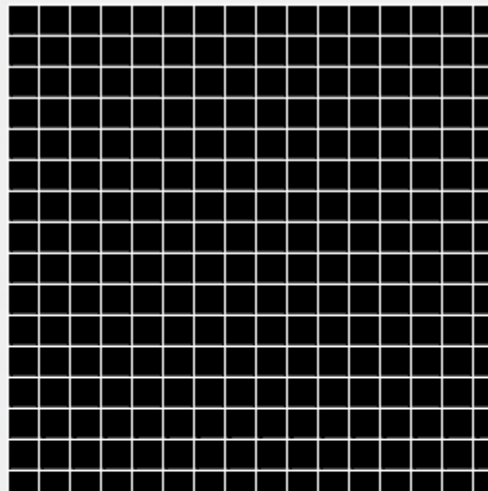
Unit Height in Pixels:

Display Width in Pixels:

Display Height in Pixels:

Base address for display:

Counter value 10



- Final cache hit rate = 75%
- Elements are accessed in same order they are stored
  - With each **MISS**, a block of 4 elements are written in
  - Next 3 accesses are **HITS** within the same cache block
  - Finally, when direct mapping maps to next cache block another **MISS**

# MARS Cache Simulator – Column Major Traversal

**Cache Organization**

Placement Policy:  Number of blocks:

Block Replacement Policy:  Cache block size (words):

Set size (blocks):  Cache size (bytes):

**Cache Performance**

Memory Access Count:  Cache Block Table: 

(block 0 at top)

Cache Hit Count:  ☐ = empty

Cache Miss Count:  ☒ = hit

Cache Hit Rate:  ☒ = miss

**Runtime Log**

☒ Enabled

**Visualizing memory reference patterns**

Show unit boundaries (grid marks): ☒ **No Spatial Locality**

Memory Words per Unit:

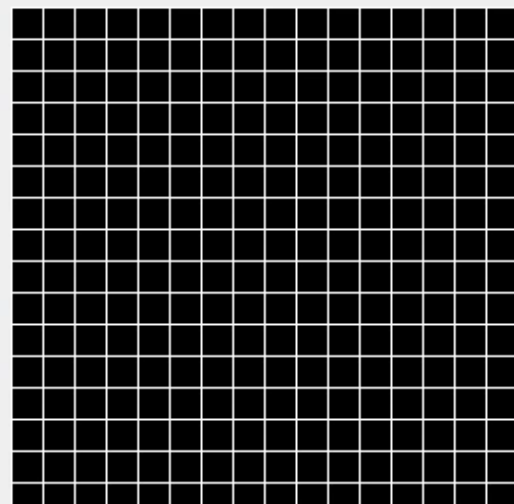
Unit Width in Pixels:


Unit Height in Pixels:

Display Width in Pixels:

Display Height in Pixels:

Base address for display:





Counter value 10

- Final cache hit rate = 0%
- Elements are not accessed sequentially
  - 16 words beyond previous access
- No two consecutive accesses occur in the same block, every access is a **MISS**
- Extreme example of non-friendly cache code

# Cache Performance Metrics



## ■ Hit Rate

- Typical percentages 95%-97% for L1 in real life

## ■ Hit Time

- Time to deliver a line in the cache to the processor
- Typical numbers 1-2 clock cycles for L1

## ■ Hit Penalty

- Additional time required due to a miss
  - Typical 50-200 cycles for main memory (trend is increasing)

- *Suppose we have a cache miss penalty of 100 cycles and a cache hit time of 1 cycle*

*Can you believe a 99% hit rate is twice as good a 97%?*

AMAT = Hit time + Miss rate x Miss penalty

99% Hits: 1 cycle + 0.01\*100 cycles = 2 cycles

98% Hits: 1 cycle + 0.02\*100 cycles = 3 cycles

97% Hits: 1 cycle + 0.03\*100 cycles = 4 cycles

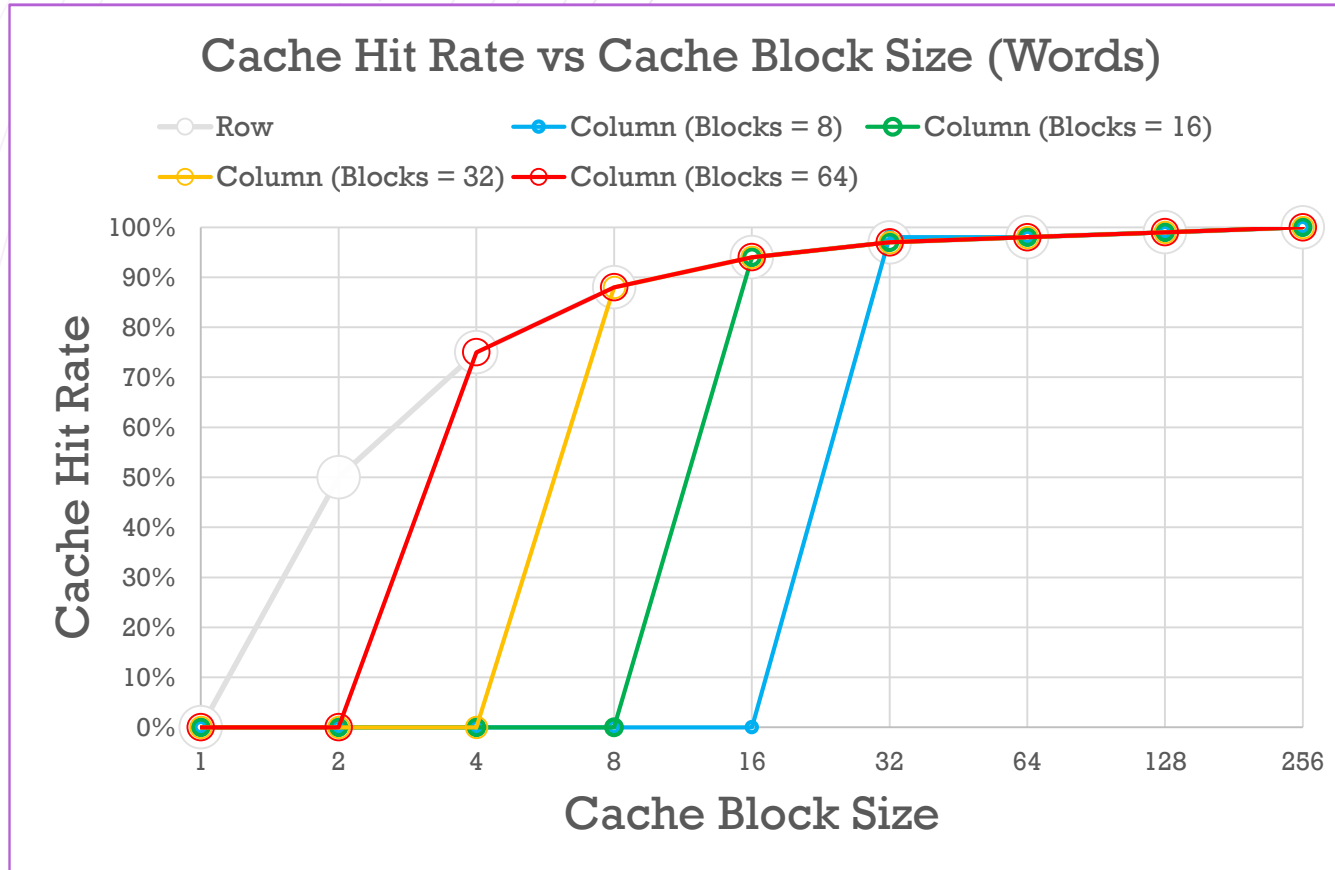
88% Hits: 1 cycle + 0.12\*100 cycles = 13 cycles

75% Hits: 1 cycle + 0.25\*100 cycles = 26 cycles

50% Hits: 1 cycle + 0.50\*100 cycles = 51 cycles

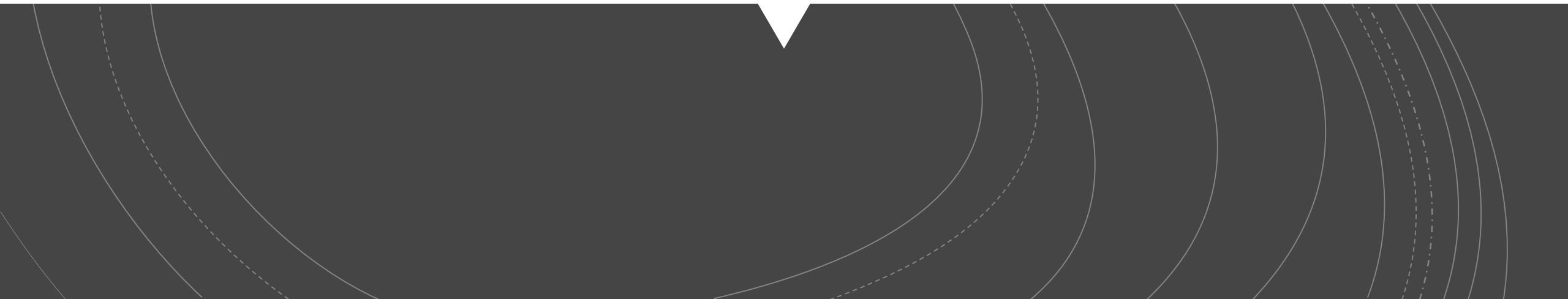


# Effects of Cache Organization



- Quantitative test: varying cache block size and observing cache hit rate
- Row traversal
  - Smooth asymptotic performance
  - Varying number of blocks does not change performance
    - Spatial locality
- Column traversal
  - Performance doesn't improve until cache block size = 32 when entire matrix fits into cache
  - Once block is read, it is never replaced

# 5. Fibonacci Two Ways



# fib(20)

- 88% cache hit rate for any fib calculation

### Simulate and illustrate data cache performance

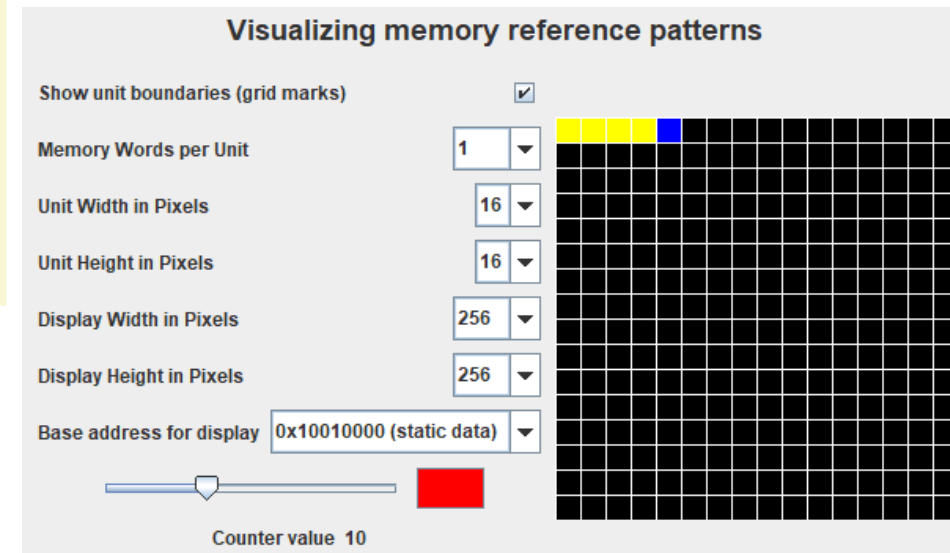
**Cache Organization**

Placement Policy	<div style="border: 1px solid black; padding: 2px;">Direct Mapping</div>		Number of blocks	<div style="border: 1px solid black; padding: 2px;">8</div>
Block Replacement Policy	<div style="border: 1px solid black; padding: 2px;">LRU</div>		Cache block size (words)	<div style="border: 1px solid black; padding: 2px;">4</div>
Set size (blocks)	<div style="border: 1px solid black; padding: 2px;">1</div>		Cache size (bytes)	<div style="border: 1px solid black; padding: 2px;">128</div>

**Cache Performance**

Memory Access Count	<div style="border: 1px solid black; padding: 2px;">17</div>	Cache Block Table (block 0 at top)  <input type="checkbox"/> = empty <input checked="" type="checkbox"/> = hit <input type="checkbox"/> = miss
Cache Hit Count	<div style="border: 1px solid black; padding: 2px;">15</div>	
Cache Miss Count	<div style="border: 1px solid black; padding: 2px;">2</div>	
Cache Hit Rate	<div style="display: flex; align-items: center;"> <div style="width: 88%; height: 20px; background-color: blue;"></div> <div style="width: 12%; height: 20px; background-color: white; border: 1px solid black;"></div> <span style="margin-left: 10px;">88%</span> </div>	



- Variables a, b, c are kept in immediate registers each iteration
- Cycle through loop repeatedly

# fib(20)

# Recursive Fibonacci

- Calculates a single Fibonacci number multiple times
- Recursive function  $T(n) = T(n-1) + T(n-2)$
- Time complexity =  $O(2^n)$
- Space complexity =  $O(n)$

```
if (n <= 1)
    return n;
return fib(n-1) + fib(n-2);
```

Simulate and illustrate data cache performance

Cache Organization

Placement Policy: Direct Mapping Number of blocks: 8

Block Replacement Policy: LRU Cache block size (words): 4

Set size (blocks): 1 Cache size (bytes): 128

Cache Performance

Memory Access Count: 0 Cache Block Table (block 0 at top)

Cache Hit Count: 0

Cache Miss Count: 0

Cache Hit Rate: 0%

☐ = empty ☒ = hit ☐ = miss

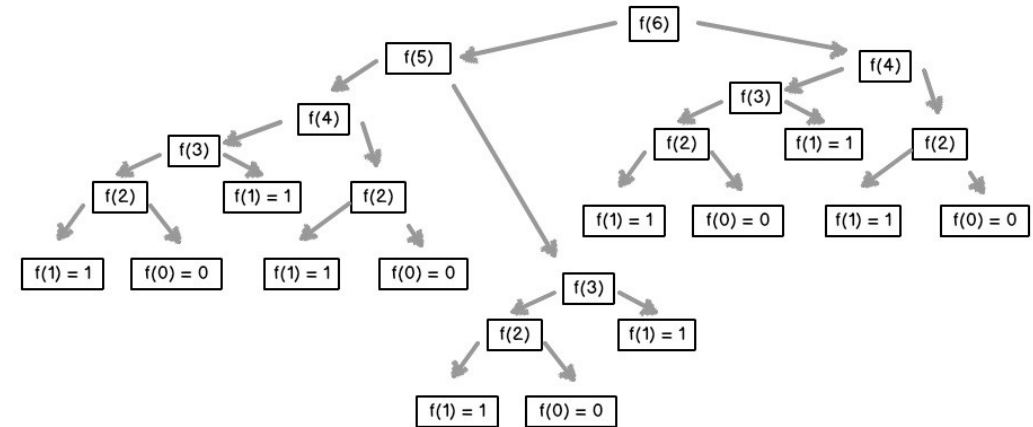
Runtime Log

☒ Enabled

*For fib(20) 367,977 instructions!*

*Exponential memory access count*

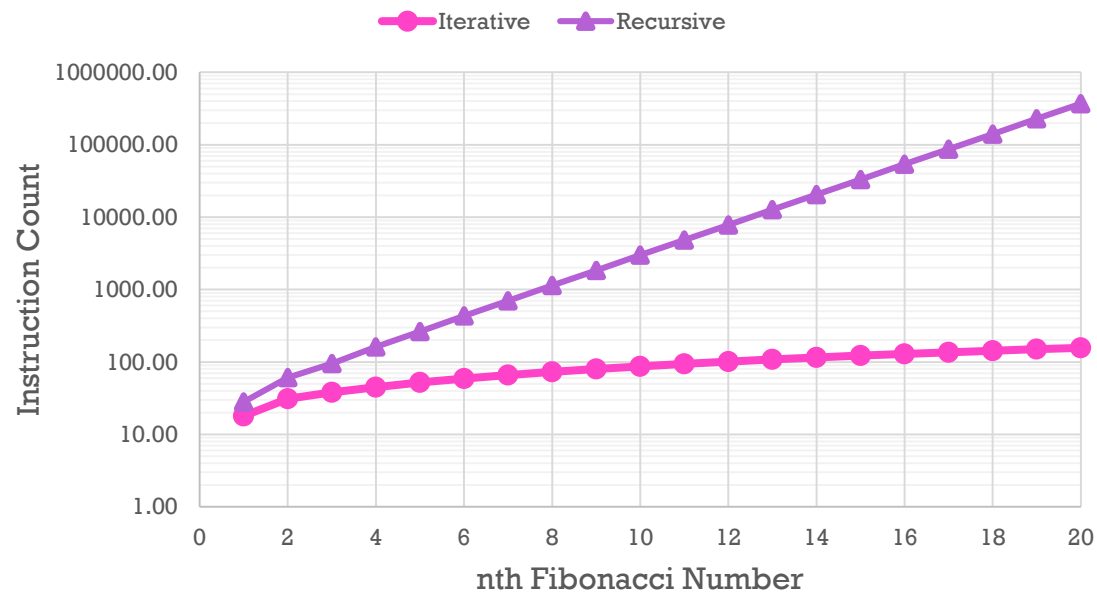
*91-100% cache hit rate for fib calculations*



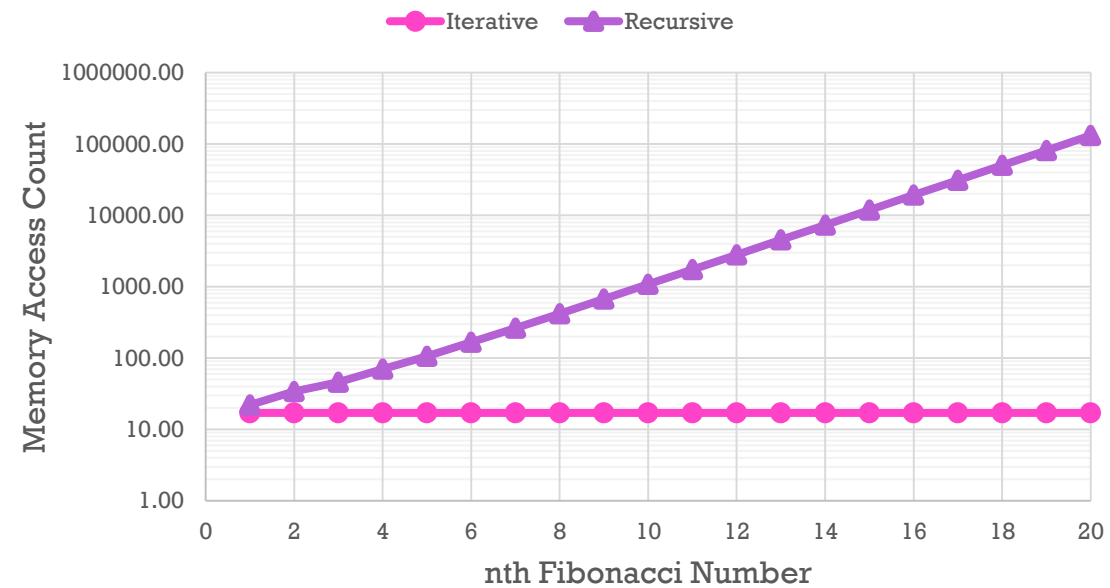
## Locality

- Working set becomes huge, follows recursion tree (temporal)
- Large strides are made between recursing back up through tree (spatial)

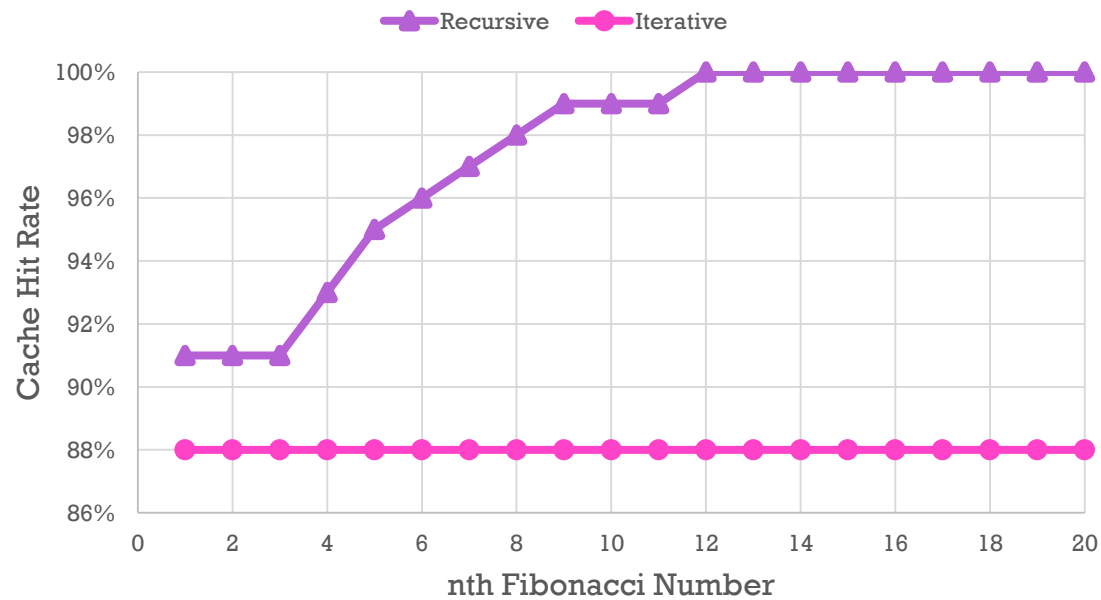
### Instruction Count



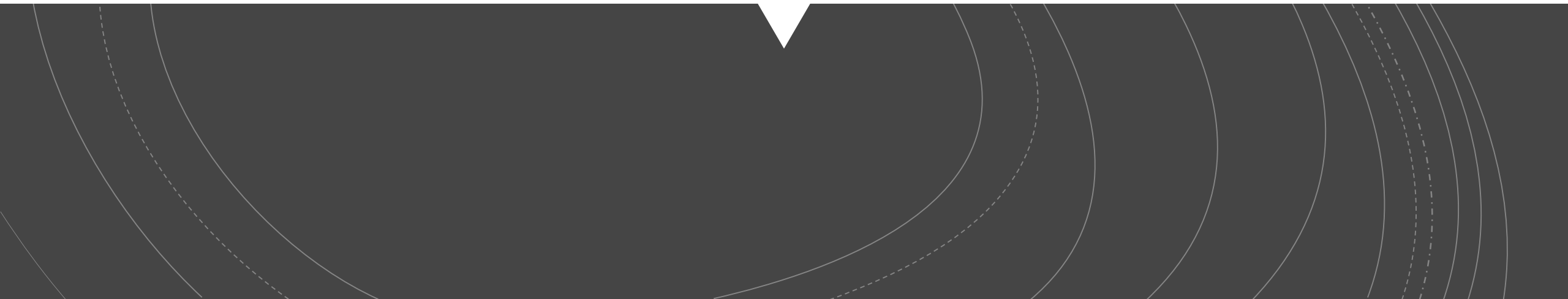
### Memory Access Count



### Cache Hit Rate



# 6. Conclusion



# Summary

- Brief overview of caches
- Observed how MIPS caching simulator works
  - Tested extreme cases of locality using matrices
- Tested recursive and iterative Fibonacci MIPS code and observed cache performance
- *There are multiple perspectives to look at for program complexity and optimization*



Write  
cache  
friendly  
code 😊

There's more to  
runtime  
performance than  
just computational  
complexity.



# References

- [1] G. Kesden and A. Rowe, "The Memory Hierarchy", Carnegie Mellon, 2011.
- [2] G. Kesden and H. Pitelka, "Cache Memories", Carnegie Mellon, 2011.
- [3] I. Raharja, "Cache Performance", King Fahd University of Petroleum and Minerals, 2018.
- [4] G. Kesden and A. Rowe, "Program Optimization", Carnegie Mellon, 2011