

# Optimizing for the Memory System

# Overview

- Memory System
- Instruction Set Support
- Code Optimizations
  
- Sources:
  - **Cortex-A72 MPCore Technical Reference Manual** (I00095\_0003\_05\_en) TRM
  - **Cortex-A72 Software Optimisation Guide** (UAN 0016A) SOG
  - **ARM Cortex-A Series Programmer's Guide** (DEN0013D) CASPG
  - **ARMv7 Architecture Reference Manual**, (DDI0406B) v7ARM
  - **ARM C-Language Extensions**, (IHI0053D) ACLE

# Great Refresher on Cache Concepts

## ARM® Cortex®-A Series

Version: 4.0

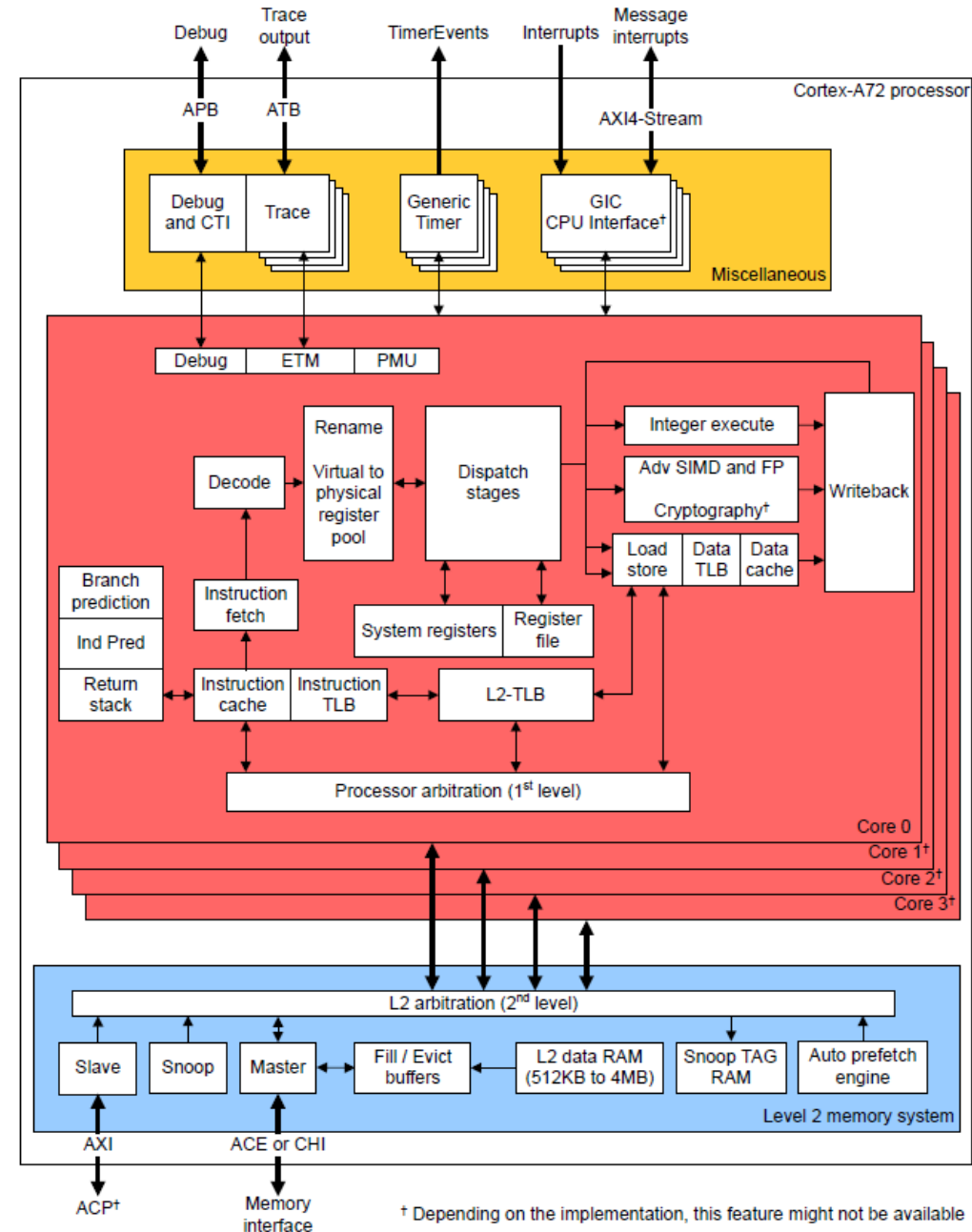
### Programmer's Guide

- CASPG, Chapter 8

- ▼ 8: Caches
  - 8.1 Why do caches help?
  - 8.2 Cache drawbacks
  - 8.3 Memory hierarchy
  - > 8.4 Cache architecture
  - > 8.5 Cache policies
  - 8.6 Write and Fetch buffers
  - 8.7 Cache performance and hit rate
  - 8.8 Invalidating and cleaning cache memory
  - > 8.9 Point of coherency and unification
  - > 8.10 Level 2 cache controller
  - 8.11 Parity and ECC in caches

# Cortex-A72 Overview

- 4 cores, each with
  - Memory Management Unit
    - 48 entry fully-associative L1 Instruction TLB
    - 32 entry fully-associative L1 Data TLB
    - 4-way set-assoc. 1024-entry L2 TLB
  - TRM Chapter 5
- Per-core L1 caches
  - 48 kB instruction cache
  - 32 kB data cache
  - TRM Chapter 6
- Shared L2 cache
  - Unified I+D cache
  - 512 kB – 4 MB
  - TRM Chapter 7



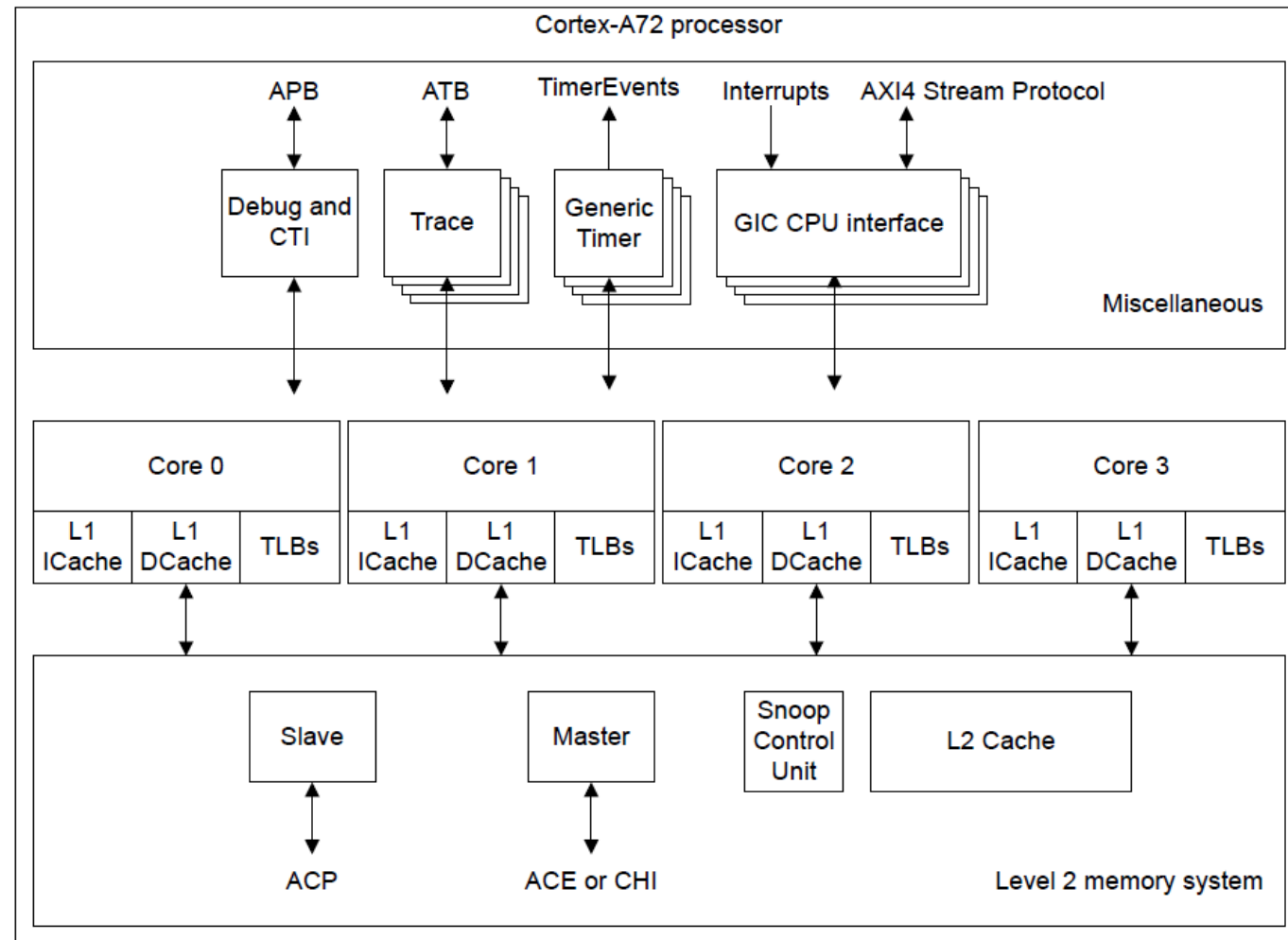
# LI Memory System

## ■ LI Instruction Cache

- 48 KB 3-way set-associative
- 64 byte line length
- 16 byte output path: 4 instructions wide
- Parity protection per halfword
- Physically indexed, physically tagged (PIPT)
- LRU replacement

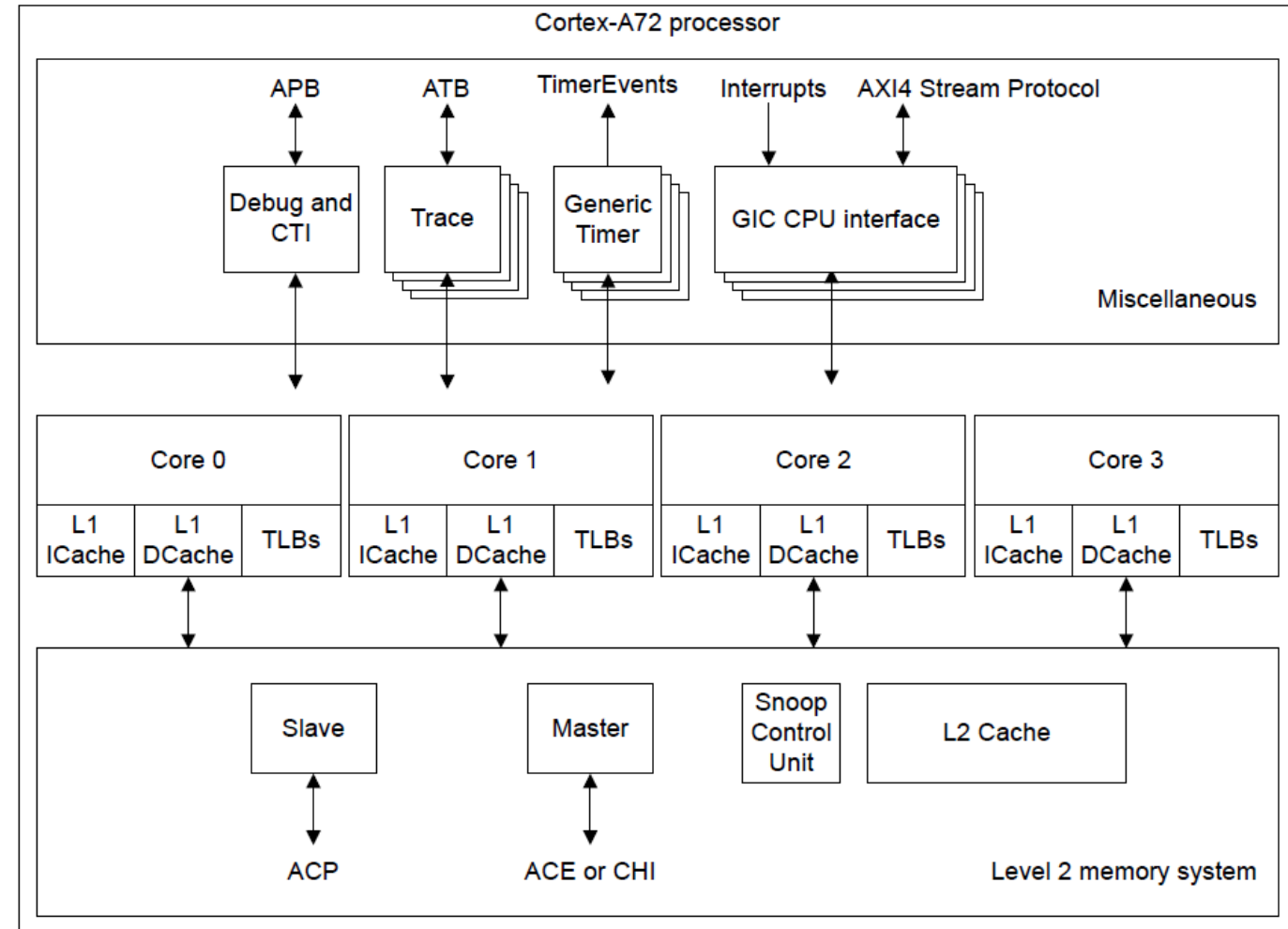
## ■ LI Data Cache

- 32 KB 2-way set associative
- 64 byte line length
- ECC per word
- PIPT, LRU
- Hardware prefetcher
- Normal memory load requests: out-of-order, speculative, non-blocking
- Device memory load requests: non-speculative, non-blocking



# L2 Memory System

- Unified cache (instructions + data)
- 512 KB – 4 MB, 16-way set-associative
- 64-byte line length
- Physically indexed, physically tagged
- Banked pipeline structures
- Programmable pseudo-LRU or pseudo-random replacement
- 20 – 28 Fill/Eviction Queues



# L2 Cache Prefetcher

- Hardware L2 prefetcher
  - Load/Store unit handles prefetch generation
  - On L2 instruction fetch, fetch consecutive cache lines (configurable for 0, 1, 2 or 3 prefetches)
  - On L2 table walk descriptor access, fetch next cache line
  - On prefetch request, forward read data before line is allocated

# Memory Access Performance

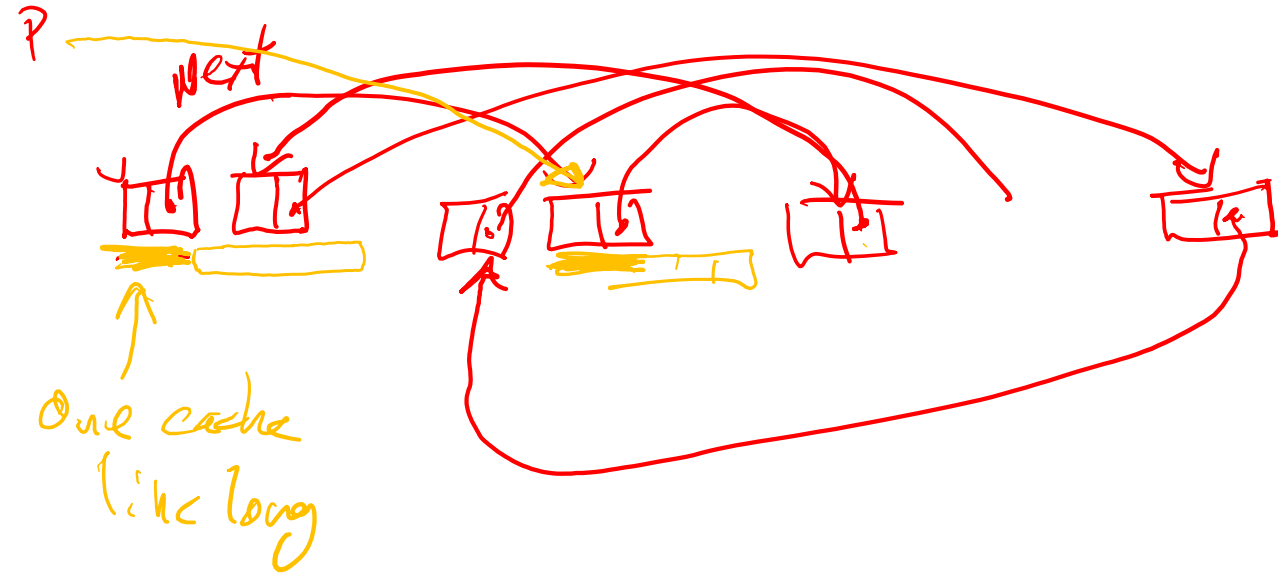
```

initialize(number_of_elements) ;
a72MeasureDataAccessEvents() ;

start_clock() ;
peStartCounting() ;
for ( ; iterations > 0 ; iterations--) {
    for (CacheLine *p = listHead ; p != NULL ;
        p = p->nextLine)
        ;
    }
peStopCounting() ;

print_clock_time(stdout, get_clock_time()) ;
a72PrintDataAccessEvents(stdout) ;

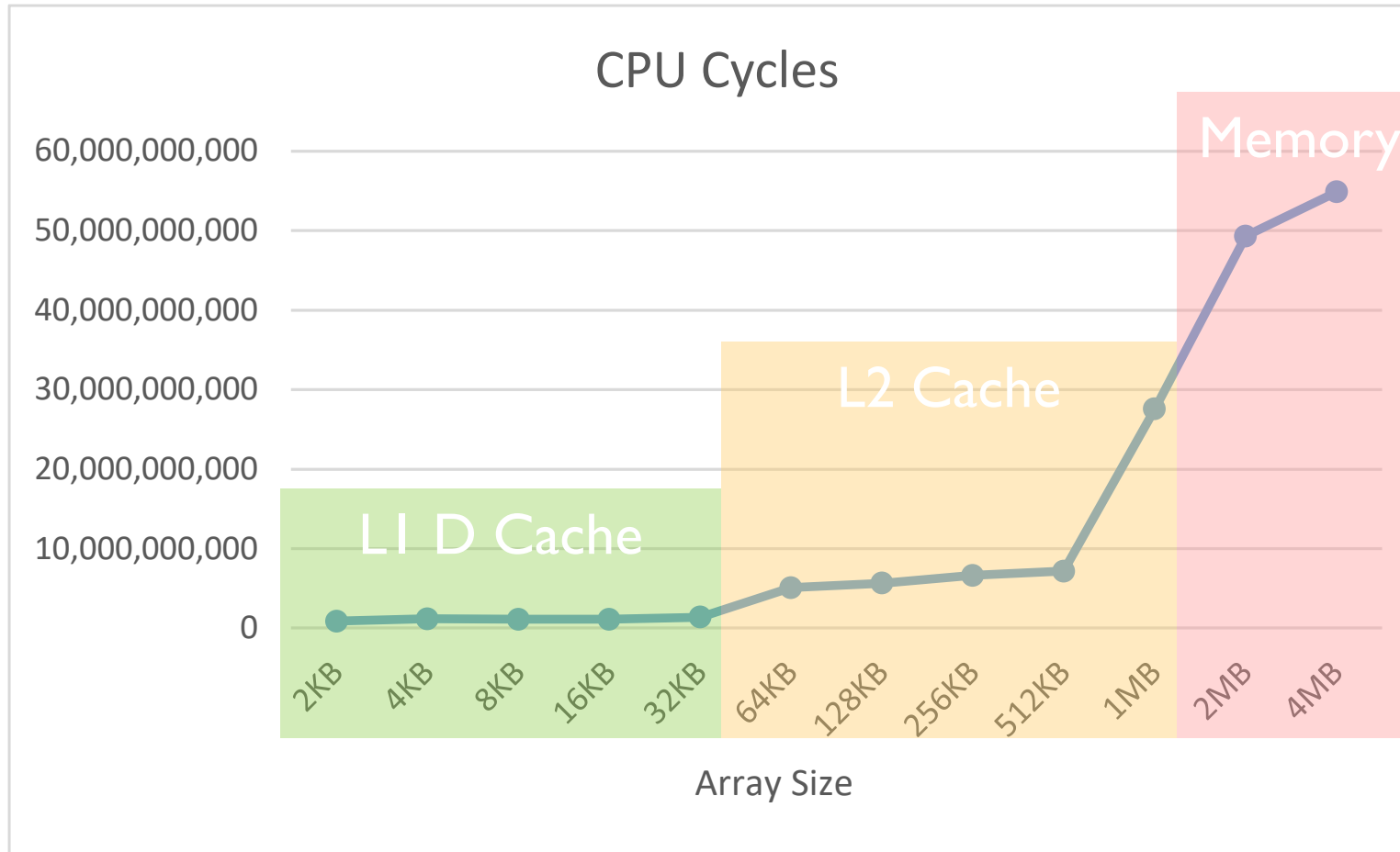
```



- From <http://sandsoftwaresound.net/arm-cortex-a72-tuning-memory-access/>
- Use pointer-chasing loop code
  - Use linked list, step through with pointer p
  - Test p. If not NULL, load p with p->nextLine.
  - Pseudorandom nextLine locations should eliminate spatial & temporal locality to make cache ineffective
- Measure execution time, PMU events
  - Each list element is 64 bytes long (takes exactly one cache line)
  - Array size and iteration counts modified so program always accesses same number of memory locations
    - # elements \* # iterations = constant = 268,435,456

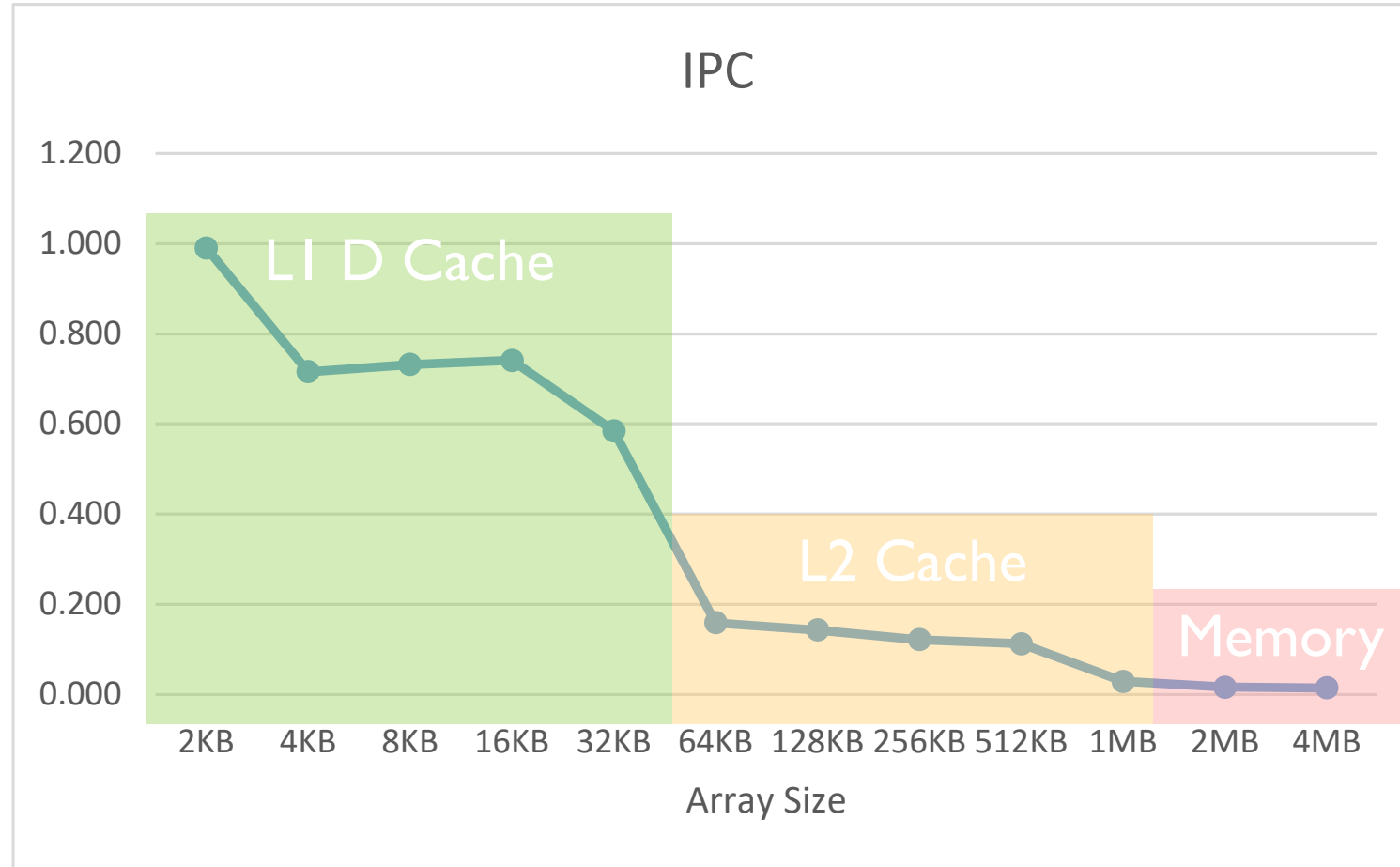


# CPU Cycles for Program Execution



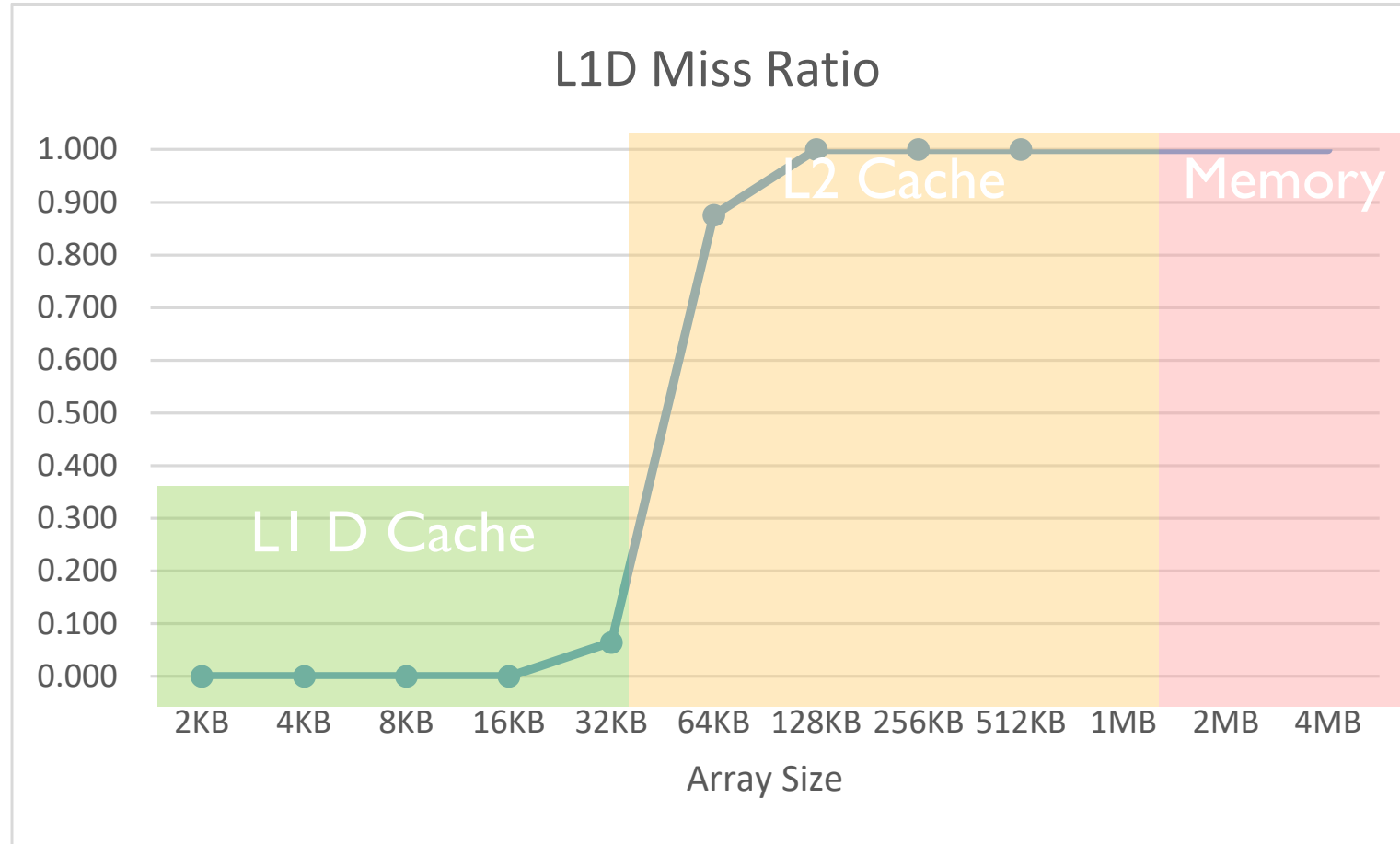
- From <http://sandsoftwaresound.net/arm-cortex-a72-tuning-memory-access/>

# Average Instruction per Cycle (IPC)



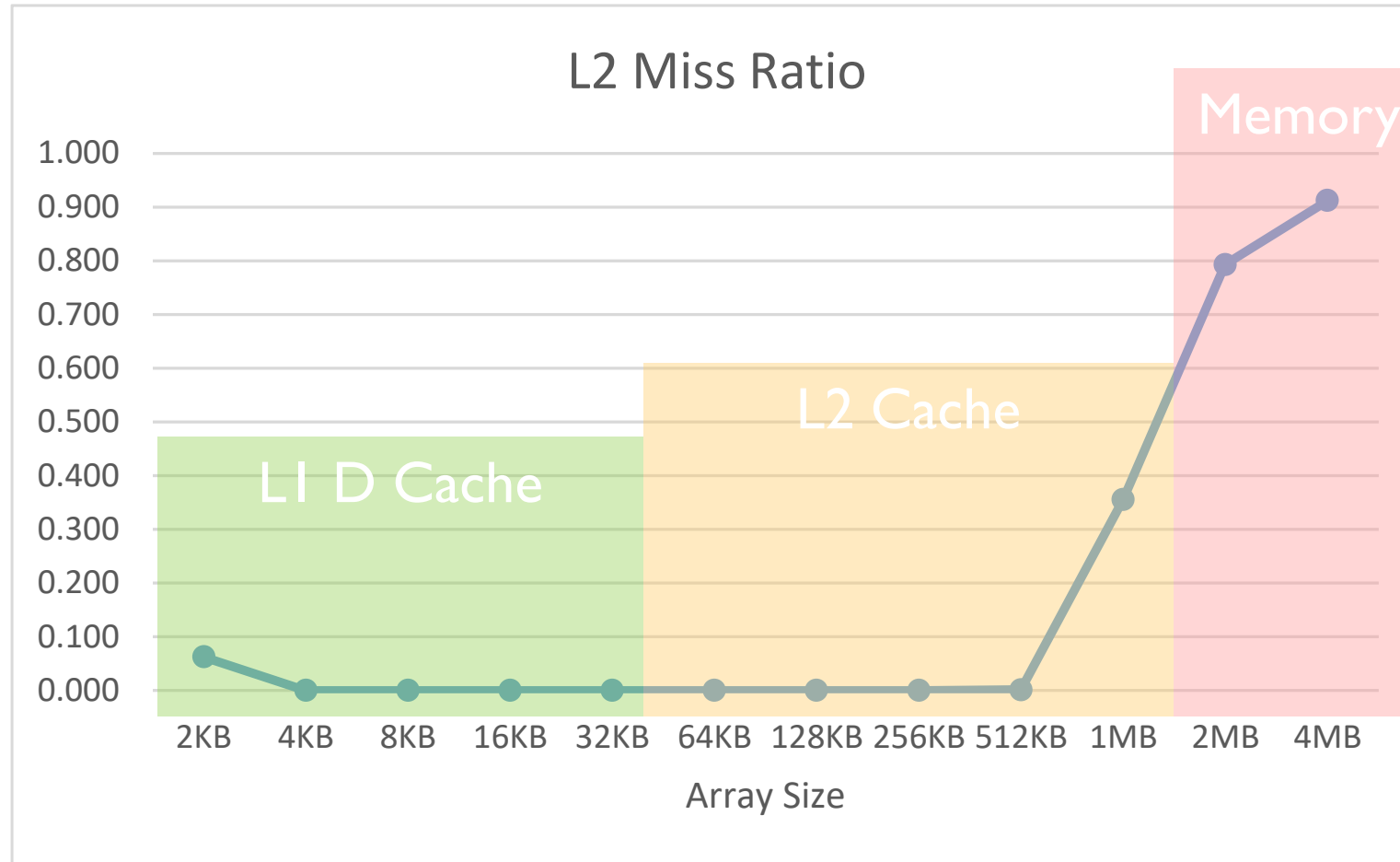
- From <http://sandsoftwaresound.net/arm-cortex-a72-tuning-memory-access/>

# LI D-Cache Miss Ratio



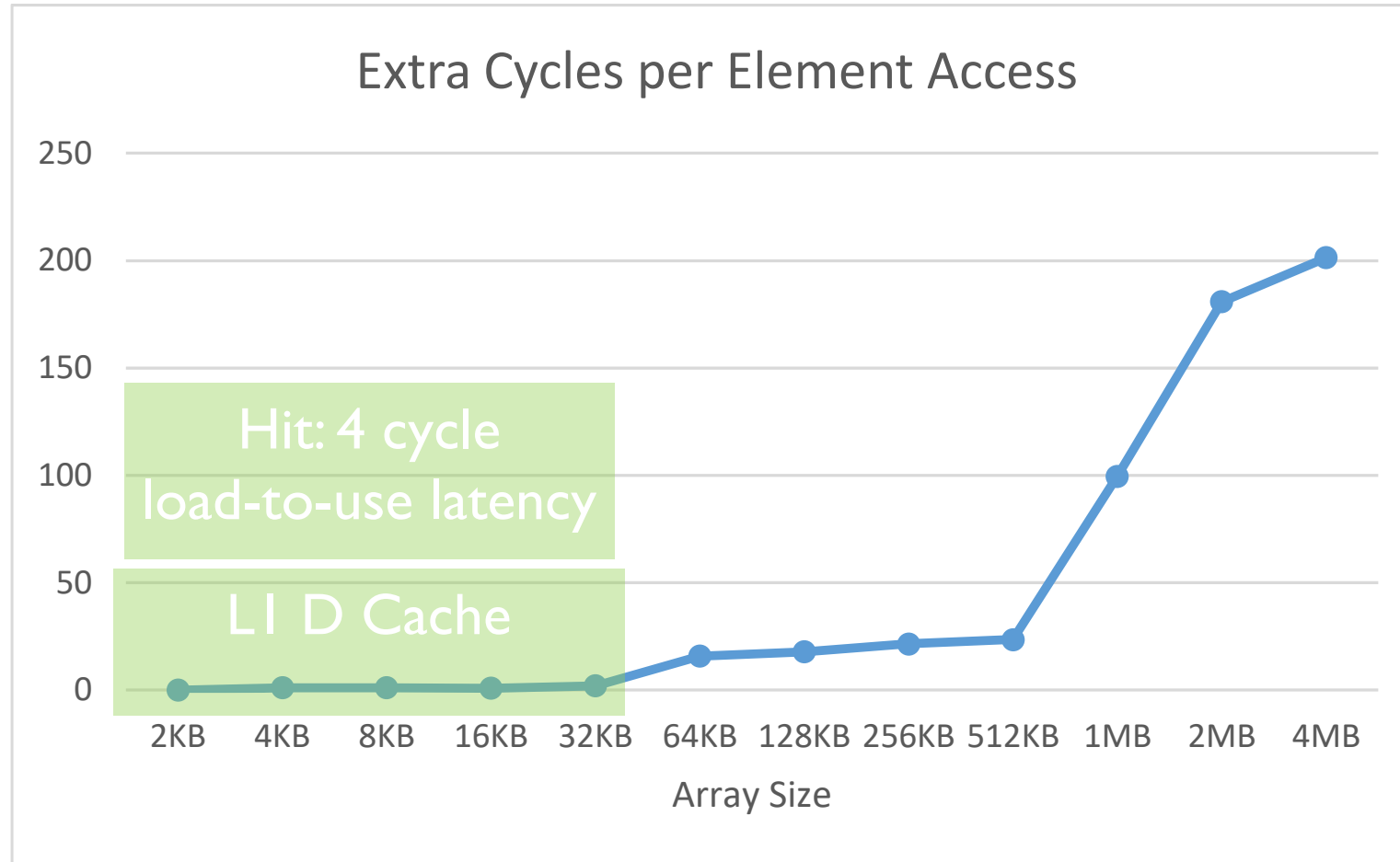
- 32 kB LI Data Cache
- From <http://sandsoftwaresound.net/arm-cortex-a72-tuning-memory-access/>

# L2 Cache Miss Ratio



- 1 MB L2 cache
- From <http://sandsoftwaresound.net/arm-cortex-a72-tuning-memory-access/>

# Extra Cycles per Element Access



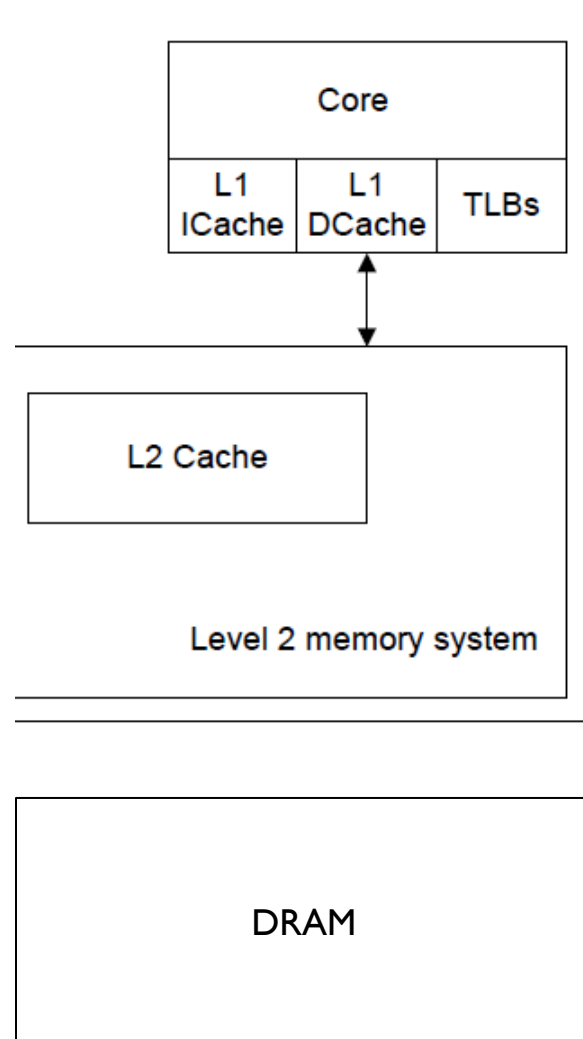
- $(\text{Total cycles for array size } X - \text{total cycles for array size 2 kB}) / \text{number of array accesses}$
- From <http://sandsoftwaresound.net/arm-cortex-a72-tuning-memory-access/> and <http://sandsoftwaresound.net/arm-cortex-a72-execution-and-load-store/>

# Test Parameters

#Elements	Iterations	Array Size	Mem Level
-----	-----	-----	-----
32	8388608	2KB	L1D cache
64	4194304	4KB	L1D cache
128	2097152	8KB	L1D cache
256	1048576	16KB	L1D cache
512	524288	32KB	L1D cache
1024	262144	64KB	L2 cache
2048	131072	128KB	L2 cache
4096	65536	256KB	L2 cache
8192	32768	512KB	L2 cache
16384	16384	1MB	L2 cache
32768	8192	2MB	RAM
65536	4096	4MB	RAM

# Instruction Set Support

# Prefetch/Preload Instructions



- Instructions which *try* to preload a data item or an instruction closer in the memory system
- Problems with using a regular load instruction to prefetch
  - Increases register use (register pressure), reducing effectiveness of compiler register allocation
  - With virtual memory, fetch from non-resident page will cause page fault, slowing program execution (unnecessarily, if prefetch was not needed)
  - How to prefetch an instruction?
- ARM ISA provides prefetch instructions for both instructions and data
  - No target register needed – just specify memory address
- Prefetch instructions will not
  - Cause synchronous aborts (e.g. page fault)
  - Change memory, cache or TLB differently than the equivalent load, store or fetch would



# Data Preload: PLD, PLDW

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

On an architecture variant that includes both the PLD and PLDW instructions, the PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write.

- PLD: load
- PLDW: store
- Different forms based on how address is provided
  - Immediate
  - Literal
  - Register
- Details in v7ARM, A8.6.117-119

PLD{W}<C><q> [ <i>&lt;Rn&gt;</i> {, #+/-<imm>}]					
W	If specified, selects PLDW, encoded as W = 1 in Thumb encodings and R = 0 in ARM encodings. If omitted, selects PLD, encoded as W = 0 in Thumb encodings and R = 1 in ARM encodings.				
<C><q>	See <i>Standard assembler syntax fields</i> on page A8-7. An ARM PLD or PLDW instruction must be unconditional.				
<Rn>	The base register. The SP can be used. For PC use in the PLD instruction, see <i>PLD (literal)</i> on page A8-238.				
+/-	Is + or omitted to indicate that the immediate offset is added to the base register value (add == TRUE), or – to indicate that the offset is to be subtracted (add == FALSE). Different instructions are generated for #0 and #-0.				
<imm>	The immediate offset used to form the address. This offset can be omitted, meaning an offset of 0. Values are: <table data-bbox="1268 1296 2007 1389"> <tr> <td><b>Encoding T1, A1</b></td><td>any value in the range 0-4095</td></tr> <tr> <td><b>Encoding T2</b></td><td>any value in the range 0-255.</td></tr> </table>	<b>Encoding T1, A1</b>	any value in the range 0-4095	<b>Encoding T2</b>	any value in the range 0-255.
<b>Encoding T1, A1</b>	any value in the range 0-4095				
<b>Encoding T2</b>	any value in the range 0-255.				

# Instruction Preload: PLI

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache. For more information, see *Behavior of Preload Data (PLD, PLDW) and Preload Instruction (PLI) with caches* on page B2-7.

- Different forms based on how address is provided

- Immediate
- Literal
- Register

- Details in v7ARM, A8.6.120-121

- Not implemented on Cortex-A72

PLI<C><q> [<Rn>, +/-<Rm> {, <shift>}]

where:

<C><q>	See <i>Standard assembler syntax fields</i> on page A8-7. An ARM PLI instruction must be unconditional.
<Rn>	Is the base register. The SP can be used.
+/-	Is + or omitted if the optionally shifted value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (permitted in ARM code only, add == FALSE).
<Rm>	Contains the offset that is optionally shifted and applied to the value of <Rn> to form the address.
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. For encoding T1, <shift> can only be omitted, encoded as imm2 = 0b00, or LSL #<imm> with <imm> = 1, 2, or 3, with <imm> encoded in imm2. For encoding A1, see <i>Shifts applied to a register</i> on page A8-10.

# Data Prefetch Intrinsic

- Source: ACLE
- Generate prefetch instruction if available (else no-op)
- Prefetch data to innermost cache level for reading
  - `void __pld(void const volatile *addr);`
- Prefetch data to given cache level for given access with given retention policy

Access Kind	Value	Summary
PLD	0	Fetch the addressed location for reading
PST	1	Fetch the addressed location for writing

Cache Level	Value	Summary
L1	0	Fetch the addressed location to L1 cache
L2	1	Fetch the addressed location to L2 cache
L3	2	Fetch the addressed location to L3 cache

```
void __pldx(
    unsigned int /*access_kind*/,
    unsigned int /*cache_level*/,
    unsigned int /*retention_policy*/,
    void const volatile *addr);
```

Retention Policy	Value	Summary
KEEP	0	Temporal fetch of the addressed location (i.e. allocate in cache normally)
STRM	1	Streaming fetch of the addressed location (i.e. memory used only once)

# Instruction Prefetch Intrinsics

- Source: ACLE
- Generate prefetch instruction if available (else no-op)
- Prefetch data to innermost cache level for reading
  - `void __pli(T addr);`
- Prefetch data to given cache level for given access with given retention policy














```
void __plx(
    unsigned int /*cache_level*/,
    unsigned int /*retention_policy*/,
    T addr);
```

Cache Level	Value	Summary
L1	0	Fetch the addressed location to L1 cache
L2	1	Fetch the addressed location to L2 cache
L3	2	Fetch the addressed location to L3 cache

Retention Policy	Value	Summary
KEEP	0	Temporal fetch of the addressed location (i.e. allocate in cache normally)
STRM	1	Streaming fetch of the addressed location (i.e. memory used only once)

# Using Prefetch Instructions/Intrinsics














- Some information here **ARM Cortex-A Series Programmer's Guide** (DEN0013D)
- Other sources available on web

- ✓  17: Optimizing Code to Run on ARM Processors
  - >  17.1 Compiler optimizations
  - ✓  17.2 ARM memory system optimization
    -  17.2.1 Data cache optimization
    -  17.2.2 Loop tiling
    -  17.2.3 Loop interchange
    -  17.2.4 Structure alignment
    -  17.2.5 Associativity effects
    -  17.2.6 Optimizing instruction cache usage
    -  17.2.7 Optimizing L2 and outer cache usage
    -  17.2.8 Optimizing TLB usage
    -  17.2.9 Data abort optimization
    -  17.2.10 Prefetching a memory block access

# Code Optimizations for the Memory System

# Read the Cortex-A Series Programmer's Guide

- **ARM Cortex-A Series Programmer's Guide (DEN0013D)**

- ✓  17: Optimizing Code to Run on ARM Processors
  - >  17.1 Compiler optimizations
  - ✓  17.2 ARM memory system optimization
    -  17.2.1 Data cache optimization
    -  17.2.2 Loop tiling
    -  17.2.3 Loop interchange
    -  17.2.4 Structure alignment
    -  17.2.5 Associativity effects
    -  17.2.6 Optimizing instruction cache usage
    -  17.2.7 Optimizing L2 and outer cache usage
    -  17.2.8 Optimizing TLB usage
    -  17.2.9 Data abort optimization
    -  17.2.10 Prefetching a memory block access

## 17.2.2 Loop Tiling

- Break loop iterations into smaller pieces
  - Array elements are reused more closely in time
- Increases locality so cache can help more
- See the non-obvious implementation details in CASPG

```
for (i = 0; i < 1024; i++)
  for (j = 0; j < 1024; j++)
    for (k = 0; k < 1024; k++)
      result[i][j] = result[i][j] + a[i][k] * b[k][j];
```

```
for (io = 0; io < 1024; io += 8)
  for (jo = 0; jo < 1024; jo += 8)
    for (ko = 0; ko < 1024; ko += 8)
      for (ii = 0, rresult = &result[io][jo],
           ra = &a[io][ko]; ii < 8;
           ii++, rresult += 1024, ra += 1024)
        for (ki = 0, rb = &b[ko][jo];
             ki < 8; ki++, rb += 1024)
          for (ji = 0; ji < 8; ji++)
            rresult[ji] += ra[ki] * rb[ji];
```

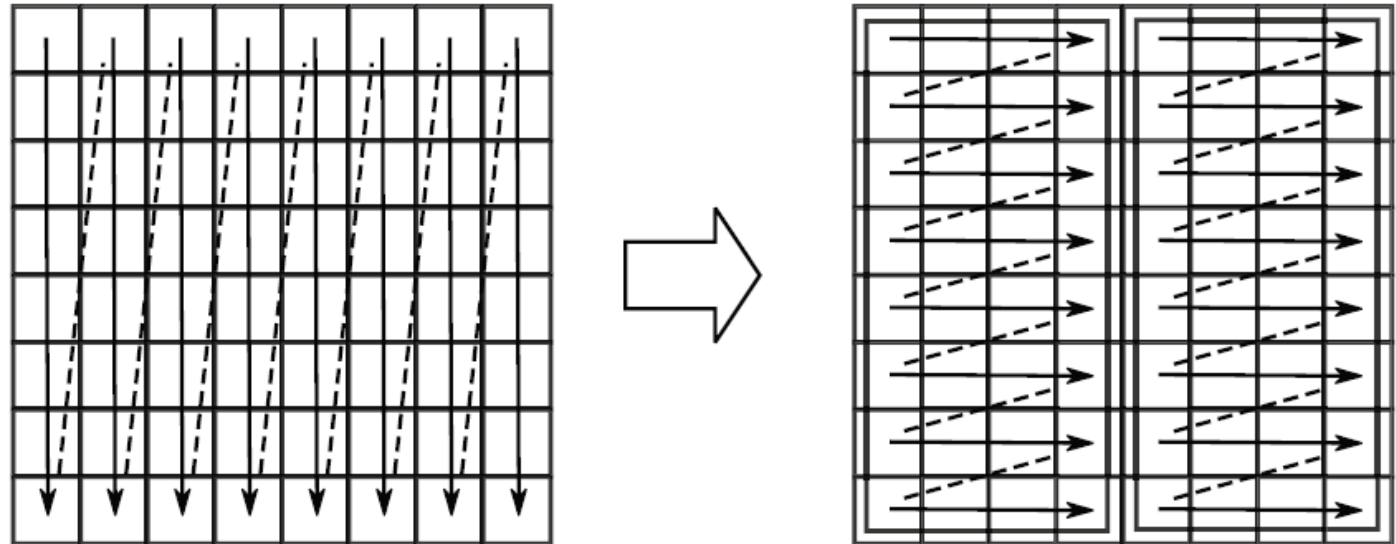


Figure 17-1 Effect of tiling on cache usage
















# More Considerations

- 17.2.3 Loop Interchange
  - Often better to re-nest loops so iteration count increases with nesting depth
    - Outermost loop: fewest iterations
    - Innermost loop: most iterations
  - Compiler support
    - GCC has `-floop-interchange`
- 17.2.4 Structure Alignment
  - Align data to not cross cache line boundaries
  - Arrange or split data structures so most-used data type fits into single cache line
- 17.2.5 Associativity Effects
  - Set-associative cache suffers if data is on boundaries of powers of 2

## And Other Good Information

- **ARM Cortex-A Series  
Programmer's Guide (DEN0013D)**

- ✓  17: Optimizing Code to Run on ARM Processors
  - >  17.1 Compiler optimizations
  - ✓  17.2 ARM memory system optimization
    -  17.2.1 Data cache optimization
    -  17.2.2 Loop tiling
    -  17.2.3 Loop interchange
    -  17.2.4 Structure alignment
    -  17.2.5 Associativity effects
    -  17.2.6 Optimizing instruction cache usage
    -  17.2.7 Optimizing L2 and outer cache usage
    -  17.2.8 Optimizing TLB usage
    -  17.2.9 Data abort optimization
    -  17.2.10 Prefetching a memory block access