Data Alignment & Padding

Optimized C++

Ed Keenan

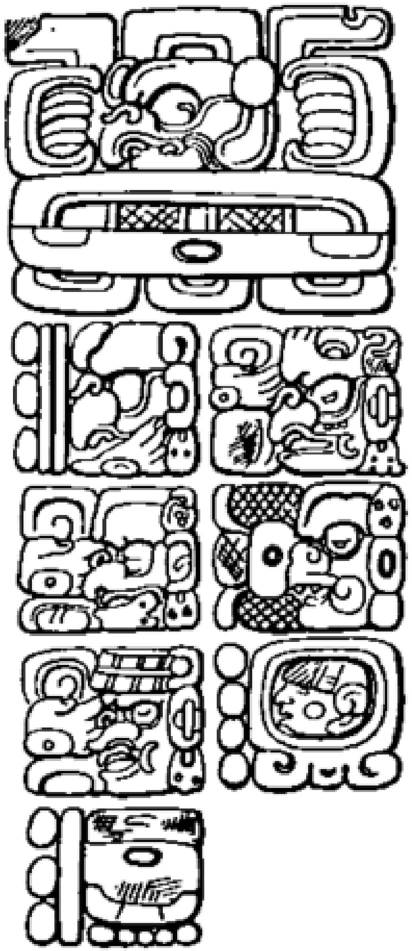
6 April 2023

13.0.10.7.18 Mayan Long Count

DePaul University

Mayan Long Count Calendar

- 1 day → *k'in*
- *20 kin* → *winal*
- 18 winals → tun
- 20 tun → k'atun
- 20 katun -> b'ak'tun
- Higher order
  - piktun, kalabtun, k'inchiltun, and alautun



DEPAUL UNIVERSITY

Mayan Confusion

- Aztec Sun Calendar is the adaptation of the Mayan Calendar

Goals

- Determine the size of data structures
- Understand the implicit padding caused by alignments
- Align data for minimize structure size
- Understanding alignment and padding as a prerequisite for caching optimizations.

Intrinsic types

- 32-bit processors
  - *char* - 1 byte
  - *bool* - 1 byte
  - *int* - 4 bytes
  - *float* - 4 bytes
  - *pointer* - 4 bytes
  - *double* - 8 bytes
- ***Warning:***
  - Data sizes are machine & compiler dependent
  - *integers* change size based on target processor
    - 64-bit vs 32-bit procs all have different base words
  - *pointers* hold the address of memory
  - *enums* are compiler dependent

sizeof()

- Unary operator *sizeof()* is used to calculate the sizes of datatypes
  - primitive types
    - Integers, floats, char
  - compound types
    - Structures, Classes, Unions
- Compile-time operator
- sizeof() calculates the complete size with data padding caused by alignments

What size is this?

```
struct A
{
    char r;      // 1 byte
    char g;      // 1 byte
    char b;      // 1 byte
};
```

sizeof(A) is 3 bytes

```
struct B
{
    int x;      // 4 bytes
    int y;      // 4 bytes
};
```

sizeof(B) is 8 bytes

Mixed data types

```
struct C
{
    int  x;      // 4 bytes
    char b;      // 1 byte
};
```

sizeof(C) is 8 not 5

- Why?

Let's look try to understand this a little more.

N-byte aligned address

- A memory address is said to be N-byte aligned when N is a power of 2 and the address is a multiple of N bytes.
- ***Confusing?***
  1. Rewrite the address in binary
  2. Count the number of least significant zeros
  3. 2 to the that number is the byte aligned.

What alignment is this address?

- 0x0036BF4: look at least significant byte (LSB):
  - 0xF4 in binary it's $1111\ 0100_b$ – it ends with 2 zeros.
  - So this address is 2^2 or 4-byte aligned
- 0x0456CF8:
  - 0xF8 in binary it's $1111\ 1000_b$ – it ends with 3 zeros.
  - So this address is 2^3 or 8-byte aligned
- 0x0456CC0:
  - 0xC0 in binary it's $1100\ 0000_b$ – it ends with 6 zeros.
  - So this address is 2^6 or 64-byte aligned

Constraints

- For optimal access
  - Data needs to be aligned to it's size.
- ***Examples:***
  - characters, bools are 1 byte
    - It's address needs to be 1-byte aligned.
  - integers, floats, pointers are 4 bytes,
    - 4-byte aligned address.
  - doubles are 8 bytes in size,
    - 8-byte aligned address

Constraints

- ***Geek Talk***
  - Processors operate in aligned native form, based on its internal registers and bus structure
  - Unaligned data, either crashes or has to the construct the data piece meal before the processor can use it.
    - This results in very slow access.
  - Check with the compiler for specific constraints
    - What works with one compiler or CPU may not be the same with another, e.g. Mips vs Intel

Back to mixed data layout

```
struct C
{
     int   x;       // 4 bytes
     char  b;       // 1 byte
};
```

Memory layout:

address				
0x0:	$x0_0$	$x0_1$	$x0_2$	$x0_3$
0x4:	b0			

- Given the alignment constraints
  - integer x, is on a 4-byte aligned address: 0x0
  - char b, is on a 1-byte aligned address: 0x4
- *What's the problem?*
  - sizeof(C) is 8 not 5
  - *Hint*: How might ***struct C*** be used in a program?

Data layout

```
struct C
{
    int  x;   // 4 bytes
    char b;   // 1 byte
};
```

```
C array_C[3];
```

array_C[0]	x0
	b0
array_C[1]	x1
	b1
array_C[2]	x2
	b2

Memory layout:

address				
0x0:	$x0_0$	$x0_1$	$x0_2$	$x0_3$
0x4:	b0	x1	x1	x1
0x8:	x1	b1	x2	x2
0xC:	x2	x2	b2	

- Alignment constraints:
  - Good:
    - x0 is 4-byte aligned
    - b0 is 1-byte aligned
  - Bad:
    - x1 is ***NOT*** 4-byte aligned

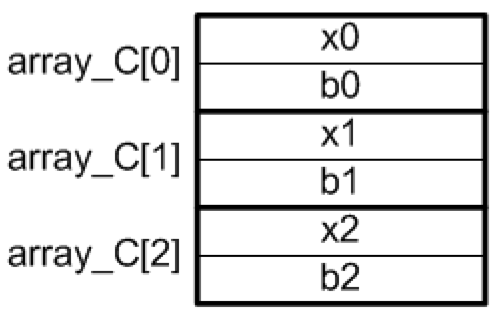
➢ Need padding to make ***x1*** aligned

Constraints

```
struct C
{
     int   x;   // 4 bytes
     char b;   // 1 byte
};


 C array_C[3];
```

address	Memory layout:			
0x0:	$x0_0$	$x0_1$	$x0_2$	$x0_3$
0x4:	b0	---	---	---
0x8:	x1	x1	x1	x1
0xC:	b1	---	---	---
0x10:	x2	x2	x2	x2
0x14:	b2	---	---	---



- Adding the padding to make ***x1*** & ***x2*** aligned to 4-bytes.
- sizeof(C) is now 8

How do you verify this?

- ***Placement new*** to the rescue
  1. Create a chuck of RAM filled with a byte data pattern, say *0xAA*
  2. Instantiate the structure on top of this data
     - Use placement new operator
  3. Initialize the data through a constructor or manually
  4. Look at the memory
     - Print it, or in a debug window

Output

```
struct C
{
    int   x;        // 4 bytes
    char  b;        // 1 byte
};

char *p = new char[128];
memset( p, 0xAA, 128);

C *r = new( p) C[3];

r[0].x = 0x90807060;
r[0].b = 0x50;
r[1].x = 0x91817161;
r[1].b = 0x51;
r[2].x = 0x92827262;
r[2].b = 0x52;
```

Memory 1

Address: r   Columns: 4

0x00369B48	60	70	80	90
0x00369B4C	50	aa	aa	aa
0x00369B50	61	71	81	91
0x00369B54	51	aa	aa	aa
0x00369B58	62	72	82	92
0x00369B5C	52	aa	aa	aa

Memory layout:

address				
0x0:	$x0_0$	$x0_1$	$x0_2$	$x0_3$
0x4:	b0	---	---	---
0x8:	x1	x1	x1	x1
0xC:	b1	---	---	---
0x10:	x2	x2	x2	x2
0x14:	b2	---	---	---

Answer to alignment question

```
struct C
{
    int  x;      // 4 bytes
    char b;      // 1 byte
};
```

- Why is struct C size 8 and not 5?
  - It pads for alignment, since the struct ***might*** be used in an array
  - Padding guarantees that the data will be aligned in contiguous memory
  - This happens at compile time

More complicated data

```
struct D
{
     void *p;   // 4 bytes
     char  a;   // 1 byte
     float x;   // 4 bytes
     char  b;   // 1 byte
     int   y;   // 4 bytes
     char  c;   // 1 byte
};
```

address	Memory layout:			
0x0:	p	p	p	p
0x4:	a	---	---	---
0x8:	x	x	x	x
0xC:	b	---	---	---
0x10:	y	y	y	y
0x14:	c	---	---	---

- sizeof(D) is 24 bytes
- A lot of padding due to the position of *x* & *y* in the structure
- Additional padding to insure that the next **p* is aligned in an array layout
- 9 bytes are consumed as padding

Make the padding obvious

```
struct D
{
      void *p;   // 4 bytes
      char  a;   // 1 byte
      float x;   // 4 bytes
      char  b;   // 1 byte
      int   y;   // 4 bytes
      char  c;   // 1 byte
};
```

- Can be rewritten to show where the padding is located
- This equivalent, doesn't change performance

```
struct D
{
      void *p;      // 4 bytes
      char  a;      // 1 byte
      char  pad1;   // 1 byte
      char  pad2;   // 1 byte
      char  pad3;   // 1 byte
      float x;      // 4 bytes
      char  b;      // 1 byte
      char  pad4;   // 1 byte
      char  pad5;   // 1 byte
      char  pad6;   // 1 byte
      int   y;      // 4 bytes
      char  c;      // 1 byte
      char  pad7;   // 1 byte
      char  pad8;   // 1 byte
      char  pad9;   // 1 byte
};
```

Rearrange data to reduce size

```
struct D
{
    void *p;     // 4 bytes
    char  a;     // 1 byte
    char  pad1;  // 1 byte
    char  pad2;  // 1 byte
    char  pad3;  // 1 byte
    float x;     // 4 bytes
    char  b;     // 1 byte
    char  pad4;  // 1 byte
    char  pad5;  // 1 byte
    char  pad6;  // 1 byte
    int   y;     // 4 bytes
    char  c;     // 1 byte
    char  pad7;  // 1 byte
    char  pad8;  // 1 byte
    char  pad9;  // 1 byte
};
```

```
struct D
{
    void *p;     // 4 bytes
    float x;     // 4 bytes
    int   y;     // 4 bytes
    char  a;     // 1 byte
    char  b;     // 1 byte
    char  c;     // 1 byte
    char  pad1;  // 1 byte
};
```

- sizeof(D) was reduced from 24 to 16 bytes with a simple rearrangement
- Only 1 byte is not used!

Guidelines for Rearrangement

- Start with the most restrictive at the top
  - Largest data alignment
- Fill in the data structure in descending byte size order
- Identify any padding with dummy variables
  - Leaves bread crumbs for future modifications
- ***Warning:***
  - Make sure before you rearrange any data structure that code base isn’t indexing to individual elements through hard coded offsets or through pointer increments

Putting it together

```
struct E
{
    char   a; // 1 byte
    double t; // 8 bytes
    float  s; // 4 bytes

};
```

- What's the size?

address	Memory layout:			
0x0:	a	---	---	---
0x4:	---	---	---	---
0x8:	t	t	t	t
0xC:	t	t	t	t
0x10:	s	s	s	s
0x14:	---	---	---	---

- Ans: 24 bytes

- Rework using our rules

```
struct E
{
    double t;      // 8 bytes
    float  s;      // 4 bytes
    char   a;      // 1 byte
    char   pad1;   // 1 byte
    char   pad2;   // 1 byte
    char   pad3;   // 1 byte
};
```

- sizeof(E) was reduced from 24 to 16 bytes

Why align data?

- You can get significant memory savings
  - Understand data layout and implicit padding
  - Rework data layout for heavily used data
- Take advantage of specialize CPU instructions
  - (128-bit, 16 byte-aligned data)
  - AVX – Advanced Vector Instruction
  - SIMD – Single instruction multiple data
  - SSE - streaming SIMD extension
  - VU - vector units
- Game systems
  - Compiler intrinsics often use ***vectors*** on XBox One and PS4 replacing assembly
  - Intrinsics on iPhone/iPad ARM processors
  - PS3 cell processors

Alignment needed for Cache!

- Trends
  - Processors are much faster than memory
  - Optimizations are dominated by memory usage
- Caches allow CPUs to reuse frequent local memory
  - Well managed cache can yield 2-4 x performance gains.
- Understanding data layouts
  - Rework data layouts according alignment constraints and the temporal use pattern

Thank You!

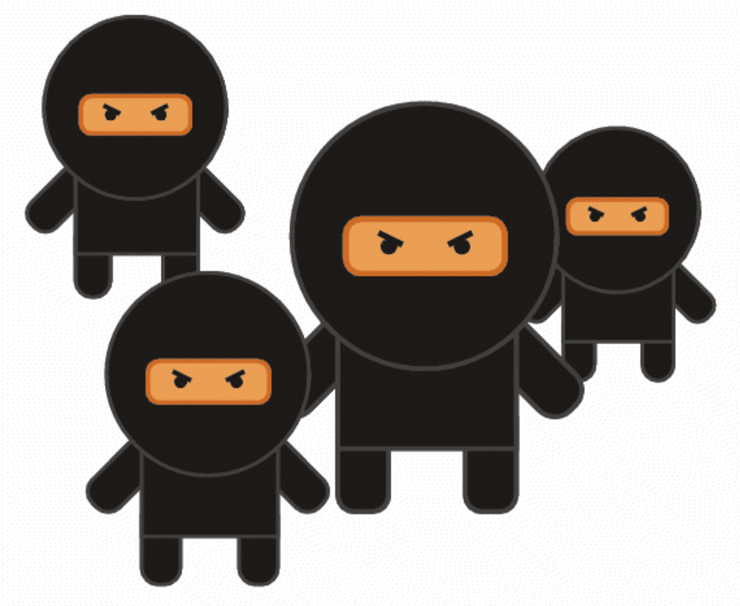
- Questions?

Data Caching

Optimized C++

Ed Keenan

Goals



- Embedded systems
- Understand High-level concepts on Data Caching
- Hardware architecture Class?
  - Very quick and arm-waving overview of Cache
  - Give you a conceptual knowledge
  - Easily 1-2 courses to understand details of CPU designs
- Best practices
  - Instruction Cache
  - Data Cache

Embedded Systems

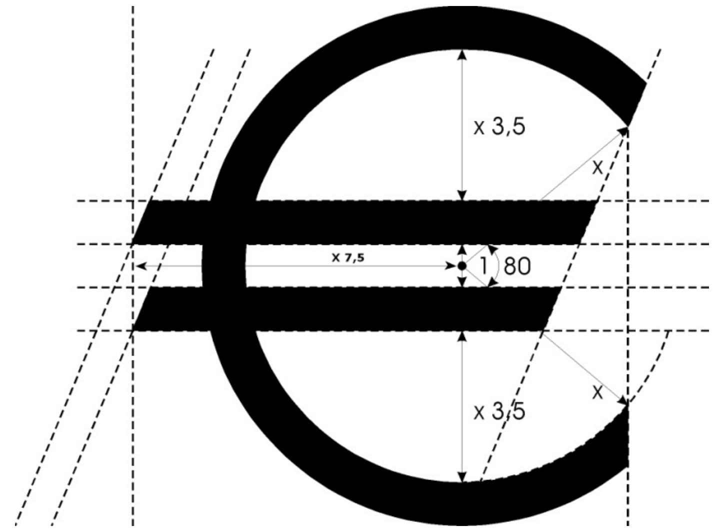
- Embedded systems have fixed resources
  - CPU
  - RAM
  - Graphics
- It's a contest between all game companies
  - Who can do more on the same hardware set?
  - Outcomes vary greatly
    - Understanding and exploiting embedded issues
      - Key to success

Upgrading Isn't possible

- Consoles are embedded systems not PCs
  - Resources are fixed
  - Upgrading isn't possible
    - Only happens on generation releases
  - Need to be optimal for every platform
    - Not the latest and fastest
      - Installed user base

Consoles

- Cost effective solutions
  - Slower
    - Buses
    - Processor speeds
    - Mass memory
  - Smaller
    - Data cache
    - Instruction cache
  - Under-clocked processors
    - Heat issues
    - Battery consumptions



Hardware is unforgiving

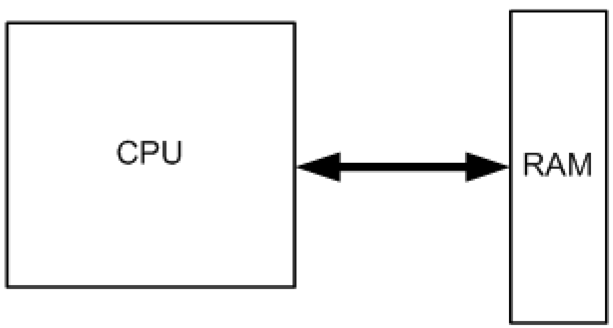
- Hardware is unforgiving for small mistakes
  - Alignment issues big penalty
    - Need to reload data
  - Data moving on and off chip
    - Large data
    - Reuse of data
  - Small data instructions cache
    - Large functions
    - Deep function stack calls

Need for Cache

- Processors are getting faster and faster
  - Memory is NOT
- Theoretical processors
  - Imagine if we had 4-5 GHz processor
    - Hitting the physical speed limit
  - All instructions could access data at this rate
  - Life would be great
- Reality
  - You would need all your RAM working at CPU rate
    - Too costly
  - Large amount of transistors and real estate

What can CPUs do?

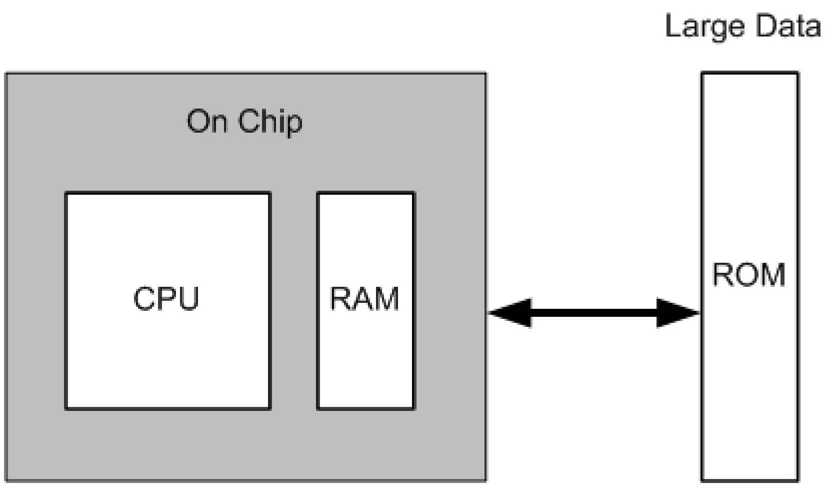
- CPUs too fast to create enough of RAM
  - Use slow cost effective RAM



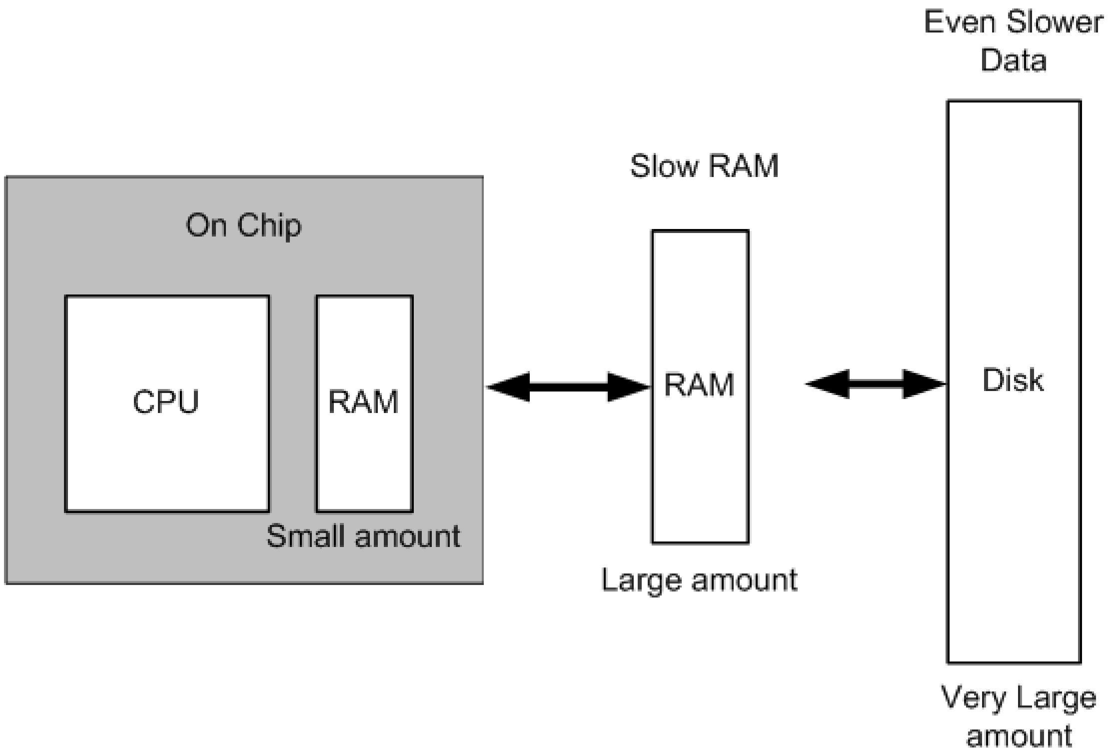
- CPU access slower RAM through wait states
  - Multiple CPU cycles to access RAM.

Early Days

- Early Days…
  - Access to bulk data was from ROM
- CPU moved data onto chip
  - Access data

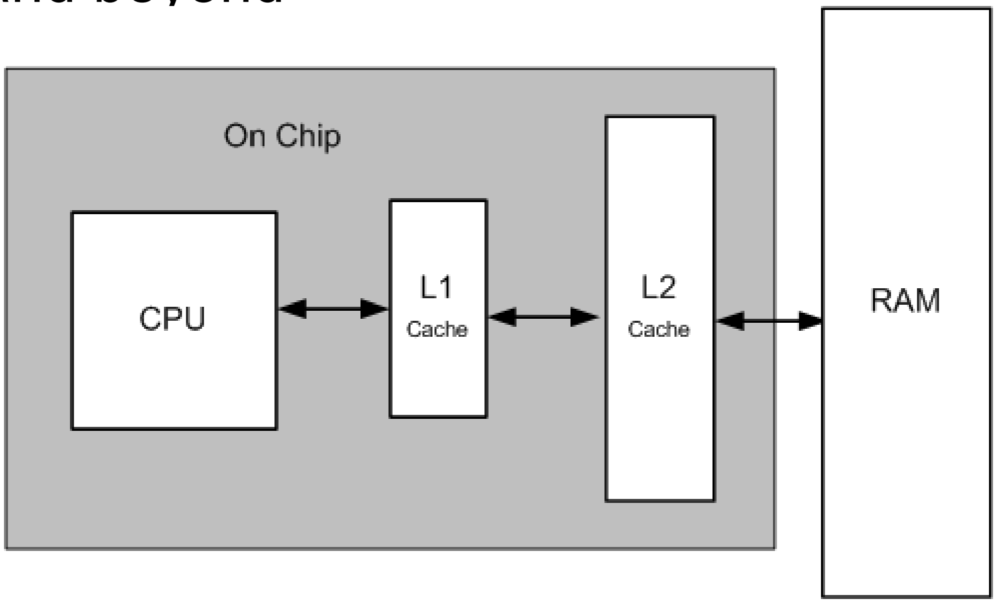


Evolution begins…



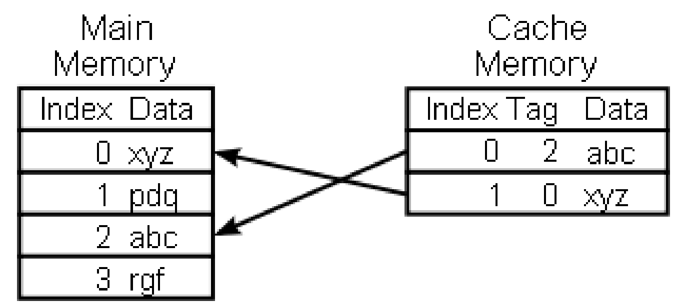
L1 - Cache

- On board Cache
  - Level 1 Cache (L1)
  - L2, L3 and beyond
- Staged storage



How do Caches work?

- Hardware saves the day.
  - Transparent to end users
  - Moves data from RAM to L2 to L1 cache
  - Magic
- Performance
  - Great for the rest of the world
  - We care where / when memory is accessed
    - We're Ninjas



I like Magic!

Cache hit

- Cache hit
  - If the Data we need is already in cache.
    - Access memory immediately
    - No additional cost, (0 cycles)
  - Cache Hit
    - Good ☺

Cache hit is the secret weapon.

Cache miss

- Cache miss
  - If the Data we need is ***NOT*** already in cache.
    - Hardware needs to get memory,
    - Transfer / move data to the cache
    - Additional cost, (Many cycles)
  - Cache Miss
    - BAD 

- Latency
  - Time it takes to get updated data into CPU access

Latency hurts!

Moving data unnecessarily sucks MORE!

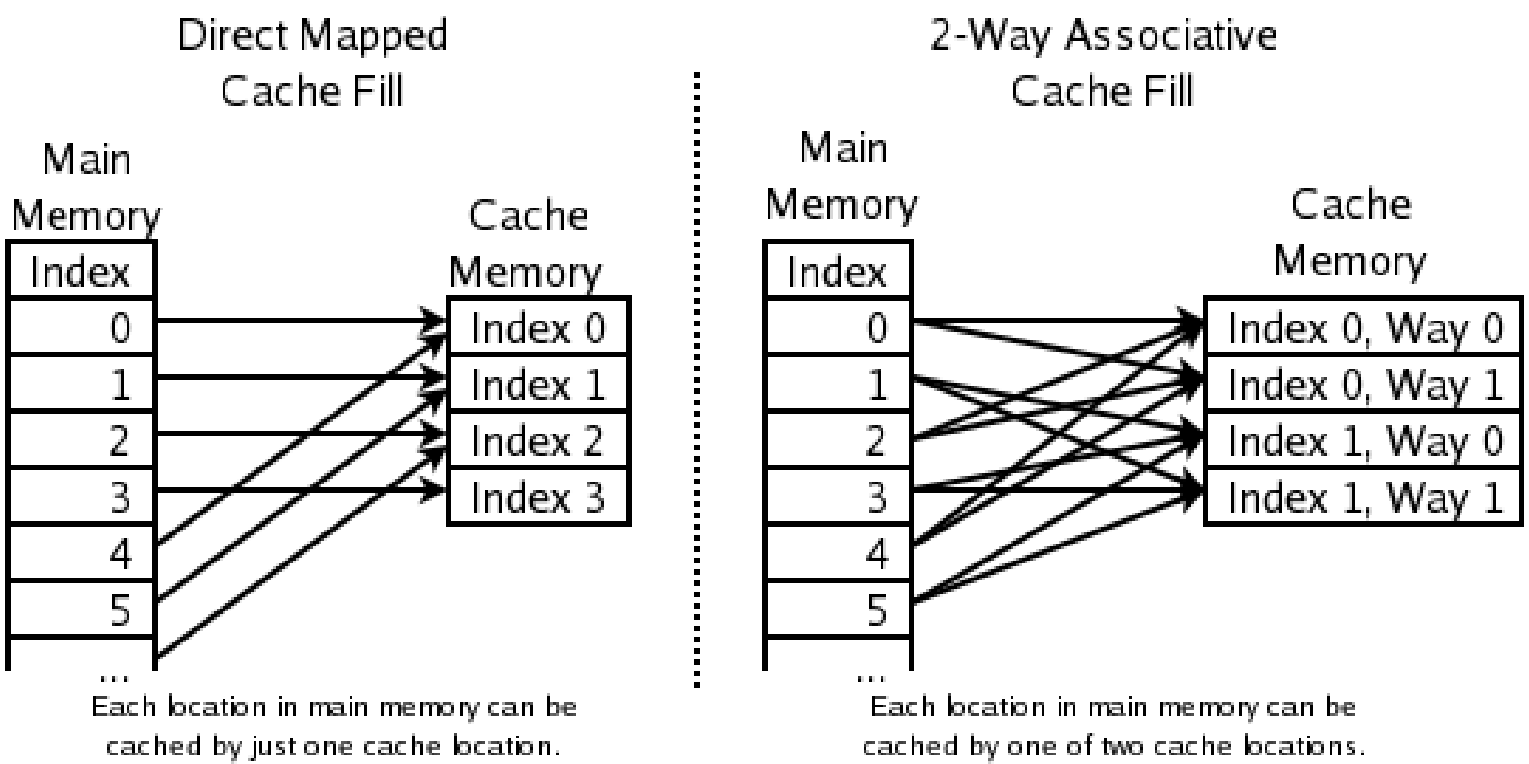
Cache Hit Ratio

- Cache Hit Ratio
  - Ratio between the average cache hit versus miss
  - Measure of efficiency on the program
    - Good programs have very good cache hit ratios
    - Naive programs do not.
  - Varies with processor / hardware layout
    - Memory layout, amount of Caches
    - Types of Caches

Caching Algorithms

- **LRU (least recently used)**
  - Discards the least recently used items first.
  - Hardware tracks age of data
    - Resets every time data is accessed
    - Through magical bits…
  - Most common (cheapest scheme)
- **MRU (most recently used)**
  - Determines a pattern where data is used frequently and intensively
  - Cyclic access patterns
  - Not used much in practice
  - But very optimal

Direct Mapped / N-way Associative



What processors to use?

- Increasing hit times and decreasing miss rates
  - Direct mapped cache
    - best (fastest) hit times
    - best tradeoff for "large" caches
  - 2-way set associative cache
  - 4-way set associative cache
  - Fully associative cache
    - Best (lowest) miss rates
    - Best tradeoff when the miss penalty is very high

Writing to Memory

- Write Through Cache
  - Every write to memory updates cache
  - Writes to main memory
    - Good – always works, no race conditions
    - Bad - Slow
- Write Back Cache
  - Uses lazy writes to update main memory
  - Lazy Writes
    - Writes to a local buffer, pools all cache
    - Doesn't write to main memory until cache is evicted.
  - ***So What?***
- ***Cache miss on Write Back Cache***
  - Need to flush the data cache back to main memory
  - Needs to reload the cache with the data
  - Blocks, waits until this is done.

Dog's Diary

Day 180

- 8:00 am - **OH BOY!**
  - DOG FOOD!
  - MY FAVORITE!
- 9:30 am - **OH BOY!**
  - A CAR RIDE!
  - MY FAVORITE!
- 9:40 am - **OH BOY!**
  - A WALK!
  - MY FAVORITE!
- 10:30 am - **OH BOY!**
  - A CAR RIDE!
  - MY FAVORITE!
- 11:30 am - **OH BOY!**
  - DOG FOOD!
  - MY FAVORITE!

- 12:00 noon - **OH BOY!**
  - THE KIDS!
  - MY FAVORITE!
- 1:00 pm - **OH BOY!**
  - THE YARD!
  - MY FAVORITE!
- 4:00 pm - **OH BOY!**
  - THE KIDS!
  - MY FAVORITE!
- 5:00 PM - **OH BOY!**
  - DOG FOOD!
  - MY FAVORITE!
- 5:30 PM - **OH BOY!**
  - MOM!
  - MY FAVORITE!

Cat’s Diary

Day 683 of my Captivity

- Today my attempt to kill my captors by weaving around their feet while they were walking almost succeeded.
  - Maybe I should try this at the top of the stairs.
- Decapitated a mouse and brought them the headless body
  - To make them aware of what I am capable of
  - Try to strike fear into their hearts.
  - They said what a good little kitty cat I was.
    - This is not working according to plan.
- There was some sort of gathering of their accomplices. I was placed in solitary confinement throughout the event.
  - I overheard that my confinement was due to my powers of inducing something called "allergies."
  - Must learn what this is and how to use it to my advantage.
- I am convinced the other captives are flunkies and maybe snitches.
  - Dog is routinely released and seems more than happy to return.
  - He is obviously a half-wit.

What’s the moral of the story?

- What can we learn from these two stories?
  - Dogs are happy that everything is working.
  - Cat is try to kill his captures and escape.
    - Trying to exploit this thing called allergies.
- What is our allergies?
  - Cache, my friend!
  - Must learn how to exploit it like the cat.

Instruction cache

- Focused on Data cache
  - Most common issues
  - Easier to control
- Instruction cache
  - Same issues and schemes
  - Caches program instructions
  - Code layout
- Number of functions and nesting play a big role
  - Can greatly effect performance
  - Playstation 2 big offender

Data cache

- Due to limited data cache
  - Data is evicted from L1 & L2
    - Quite frequently
  - Date misses
    - Causes the data to be reloaded
- Large functions that use walk-throughs
  - Read or write data in their functions
  - Evicted often

Data Cache

Solutions

- Keep data sets small
  - Tight little packages
    - Divided by use
    - Larger structures have data pointers
      - External data blocks
  - Keep like data arrange together
    - Pointers for data structures
      - Maps, Trees, tables
      - Keep together

Data Cache

Solutions

- Alignment
  - Understand any implicit data alignment
  - Identity implicit data alignment with discrete padding.
- Make sure all identified data
  - Be aware of padding
  - Rearrange to save cache issues

- class data_A

```
{
char a;     //  1 – data ,3 - pad
float x;     // 4  bytes
char b;     //  1 – data, 3 - pad
float z;     //  4 bytes
char c;     //  1 – data, 3 – pad
};
```

- Total size:
  - 20 bytes
- Rearrange:
  - floats together
  - characters together
  - Total size: 12 bytes

Thank You!

- Questions?

Hot / Cold Data Structures

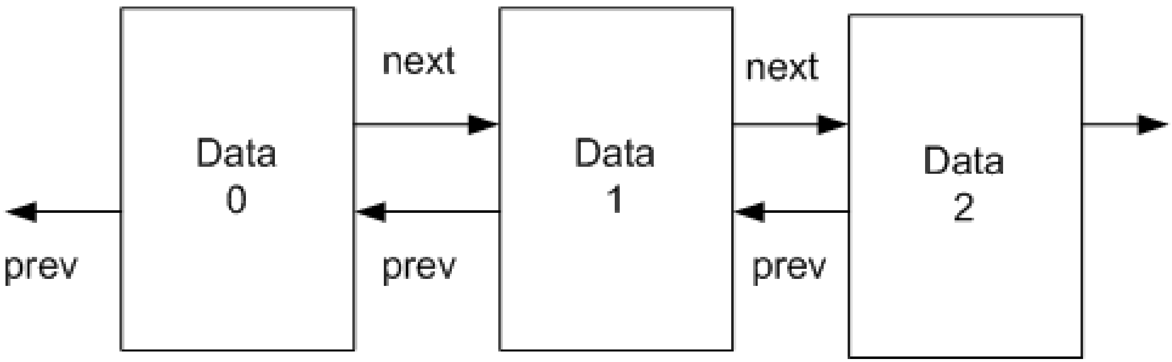
Optimized C++

Ed Keenan

Goals

- Introduce Hot / Cold data structures
- Give an intuitive feel towards data caches
- Party conversation material
  - Impress your friends
    - Give it a try

Data structure



- Example
  - Double Linked List
  - Important to understand this example 100%

```
struct data
{
     data *next;
     data *prev;
     float x;
     float y;
     float z;
     float a;
     float b;
     int   key;
};
```

Data structure

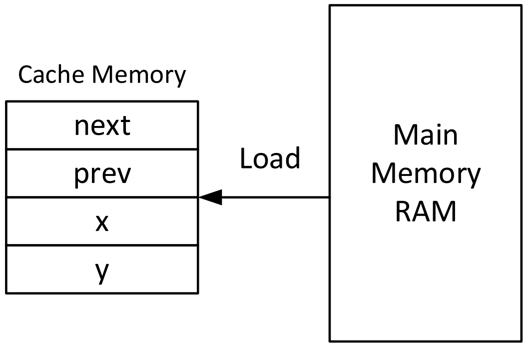
Assumptions

```
struct data
{
    data *next;
    data *prev;
    float x;
    float y;
    float z;
    float a;
    float b;
    int   key;
};
```

- What's going on?
  - Look at this from the data cache perspective
- Cache lines
  - Generally 8-DWORDs deep
- ***Let's assume for these examples***
  - single cache line
  - 4 DWORD in size.
  - Single line deep
  - 1-way associated

Let's roll…

```
struct data
{
    data *next;
    data *prev;
    float x;
    float y;
    float z;
    float a;
    float b;
    int   key;
};
```



- You start to access the data:
  p->next; --- for example
- No data is in memory
  - ***Cache Miss***
- Data is loaded into memory
  - Latency (delay) to load data
  - 1 cache line (4-DWORDS) that contains ***p->next;*** is loaded
  - Actually, ***next, prev, x, y*** variables are loaded into RAM

Access cached variables

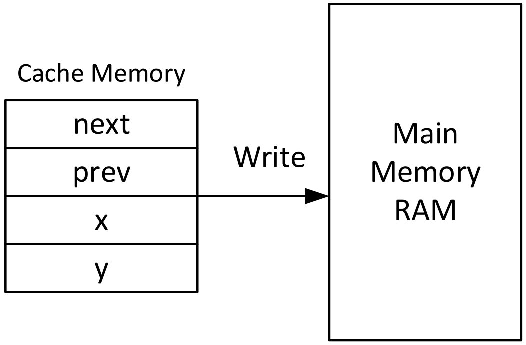
```
struct data
{
    data *next;
    data *prev;
    float x;
    float y;
    float z;
    float a;
    float b;
    int   key;
};
```

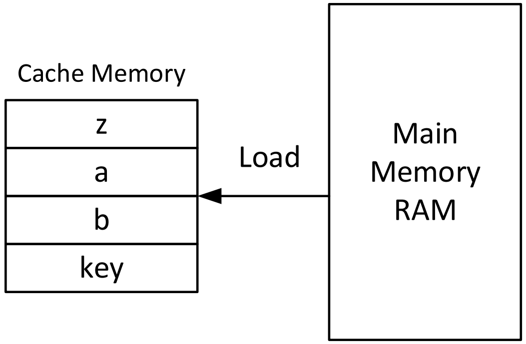
Cache Memory
next
prev
x
y

- Now we can access actually any one of those variables quickly and without delay.
  - Access, ***next, prev, x, y*** freely.
  - It's in RAM
- Now we want to read the ***p->key***.
  - What happens?
  - It's not in cache
    - ***Cache Miss***

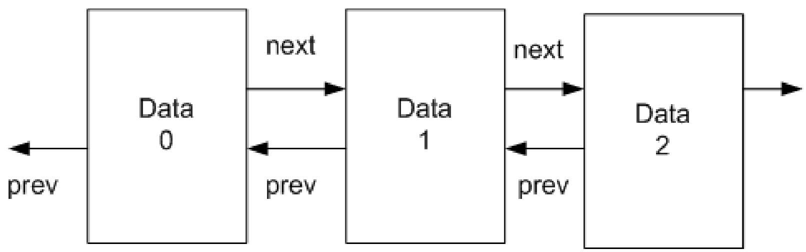
Load new data

- 1^{st} evict current cache.
  - If data was updated, the data is written to main memory.
  - If not, it's just dropped.
- Loads the new cache line that contains the ***key*** variable.
  - It actually loads, ***z, a, b, key*** into the cache.





Look at Key




```
struct data
{
    data *next;
    data *prev;
    float x;
    float y;
    float z;
    float a;
    float b;
    int   key;
};
```

Cache Memory

z
a
b
key

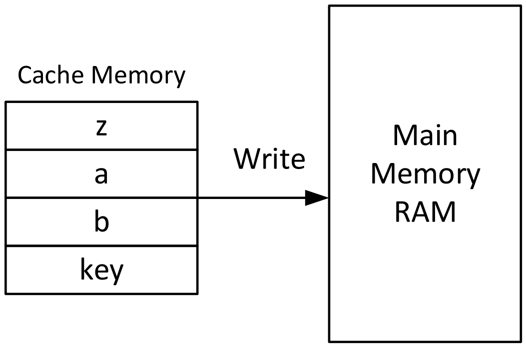
- Now that ***key*** is in cache
  - Can read it
  - Can use that variable.
- What if it's the wrong "***key***"?
  - Need to go to the next data structure.
  - Need to dereference the ***p->next*** pointer.
    - It's not in Cache

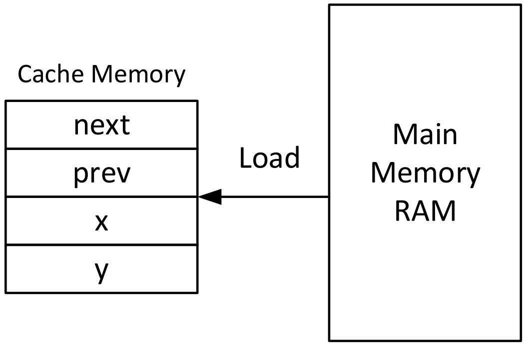
Shit!

DePaul University

Go to next pointer

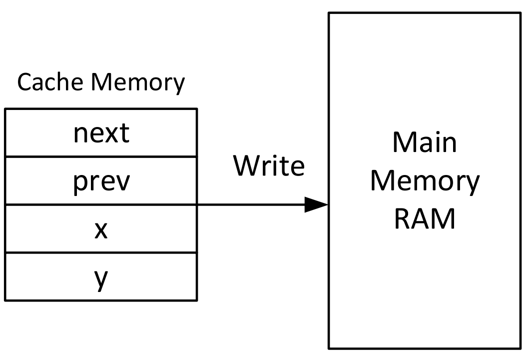
- Load the ***next*** pointer ***AGAIN***.
  - Evict the current cache line
- Load the cache line that contains the next pointer.
  - Delay… it takes time
    - Great this is the data we had before…
    - If I only knew I needed it.
- Set the new pointer to that one.
  - ***p=p->next;***



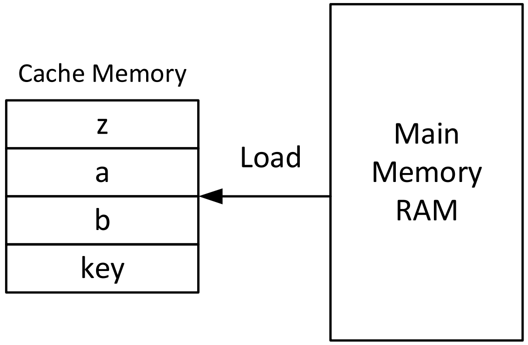


Next node

- Access the ***key*** of the 2nd data node.
- But it's not in memory
  - Crap!
  - Here we go again…
- Evict cache



- Load new cache line



- Now we have the ***key*** of the 2nd data node.

You get the drill

- Rinse an repeat…
  - We are getting beat up on cached data.
- Obviously this exaggerated
  - Since I limiting the discussion to a single deep, 4 DWORD cache.
  - But it happens.
- Can we do better?
  - Yes
  - We understand alignment and data organization
    - We can exploit that
  - We have basic understanding of data caches
    - We can exploit that too

Rework data structure

- Rework the data layout to reflect our use pattern.
  - We tend to use the next pointer and the key together.
  - Why not move them physically near each other.

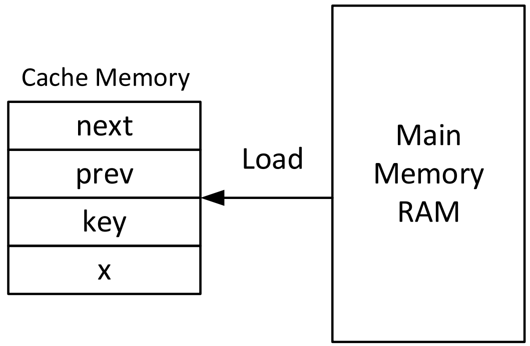
```
struct data
{
    data *next;
    data *prev;
    float x;
    float y;
    float z;
    float a;
    float b;
    int   key;
};
```

→

```
struct data
{
    data *next;
    data *prev;
    int   key;
    float x;
    float y;
    float z;
    float a;
    float b;
};
```

Load Key

- Let's look at our use pattern.
  - Load the ***key***.
    - ***p->key***;
  - Loads the cache line that contains, key.
    - You get ***next, prev, key, x*** in the same cache line.
    - That's Good




```
struct data
{
    data *next;
    data *prev;
    int   key;
    float x;
    float y;
    float z;
    float a;
    float b;
};
```

Free data grab

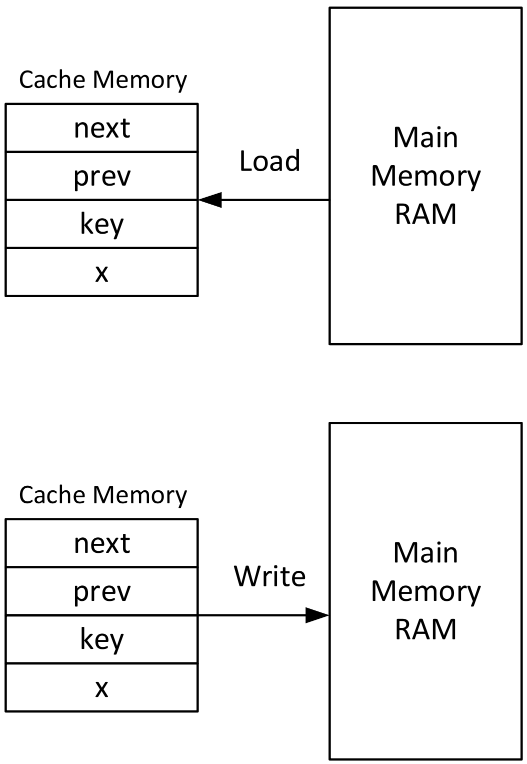
- Say it's not the correct key.
  - Need to go to the ***next*** data structure.
  - Wait the ***next*** pointer,
    - "that's in cache!"
- Ladies and Gentlemen
  - I introduce the
    - ***Cache Hit***
  - Yeah!
    - (Crowd goes wild)

```
struct data
{
     data *next;
     data *prev;
     int   key;
     float x;
     float y;
     float z;
     float a;
     float b;
};
```

Cache Memory

next
prev
key
x

Advance to next node



- Now go to the next node.
  - Load the cache line
  - Get the ***key*** and ***next*** variable with on cache read.
- Rinse and repeat.

What did we learn so far

- Good stuff
  - Keep like variables together in memory layout
  - Not understanding caching hurts performance.
  - Change data layout doesn’t change the executing code
  - Say this 100 times before you go to bed.
    - I like Cache Hits
    - Cache hits speeds up my code
    - Cache misses make you weak.
- Hazards
  - Do not use hard addressing,
    - Like pointer offsets, *(p+1) very data dependent.

Say 100 times

- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits
- I like Cache Hits

Say 100 times

- Cache hits speeds up my code
- Cache hits speeds up my code
- Cache hits speeds up my code
- Cache hits speeds up my code
- Cache hits speeds up my code
- Cache hits speeds up my code
- Cache hits speeds up my code
- Cache hits speeds up my code

Say 100 times

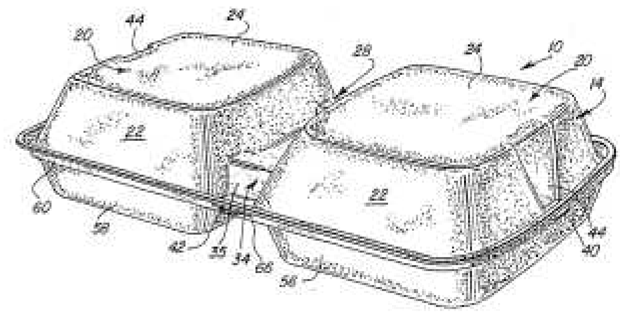
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!
- Cache misses make you weak!

Hazards!

- Do not use hard addressing,
  - Like pointer offsets, *(p+1) very data dependent.
- Why is that bad?
  - Breaks existing code once you rearrange the data

Hot / Cold data structures

- As you can see,
  - Data layout is king!
  - Too often we have huge bloated structures
- What if we rearrange the data into fast access and slow access.
  - What I call:
    - ***Hot / Cold data structures***
  - Influenced from the famous McDLT



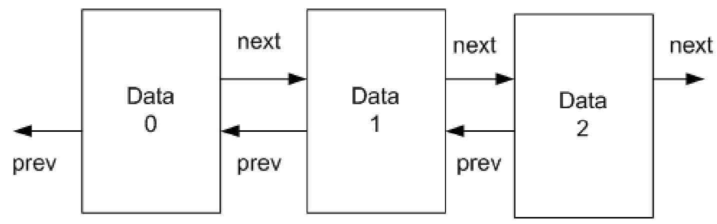
- The McDLT was sold in a novel form of packaging where the meat and bottom half of the bun was prepared separately from the lettuce, tomato, American cheese, pickles, sauces, and top half of the bun and both were then packaged into a specially designed two-sided container.
- The consumer was then expected to finalize preparation of the sandwich by combining the hot and cool sides just prior to eating.

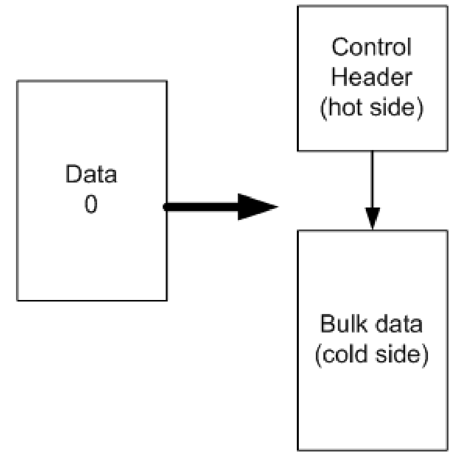
Technical Video

- Quiet please
- I need your full and undivided attention

What are we doing

- Say you have some data structure
  - Link lists, maps, queues, trees, etc…
- The data is very large, it's a fat bloated structure.
- We will ***refactor*** the code and data to exploit cache friendly schemes
  - Based on use patterns.





What to divide

- The ***Hot side***, will be the ***next, prev***, and other tree transversal structures PLUS any necessary ***key*** or ***ids***.
- The ***Cold side*** is the complete bloated data.

- Why it works.
  - The general pattern is searching or moving data within a structure.
- Use small, cache friendly data structures ***(Hot side)***
  - To quickly search, insert, replace, or walk nodes.
  - Once you found the node.
- Jump to the complete data structure ***(Cold side)***
  - To read, write complete nodes, might not be cache friendly
- Very fast in practice!

Convert original structure

```
struct orig_data
{
    data   *next;
    data   *prev;
    vect    dir;
    vect    vel;
    vect    accl;
    vect    pos;
    float   height;
    float   mass;
    int     state;
    float   a;
    float   b;
    matrix  world;
    char    name[64];
    int     key;
};
```

```
struct header_data
{
    data   *next;
    data   *prev;
    int     key;
    big_t  *cold;
};

struct big_t
{
    header_data *header;
    vect         dir;
    vect         vel;
    vect         accl;
    vect         pos;
    float        height;
    float        mass;
    int          state;
    float        a;
    float        b;
    matrix       world;
    char         name[64];
};
```

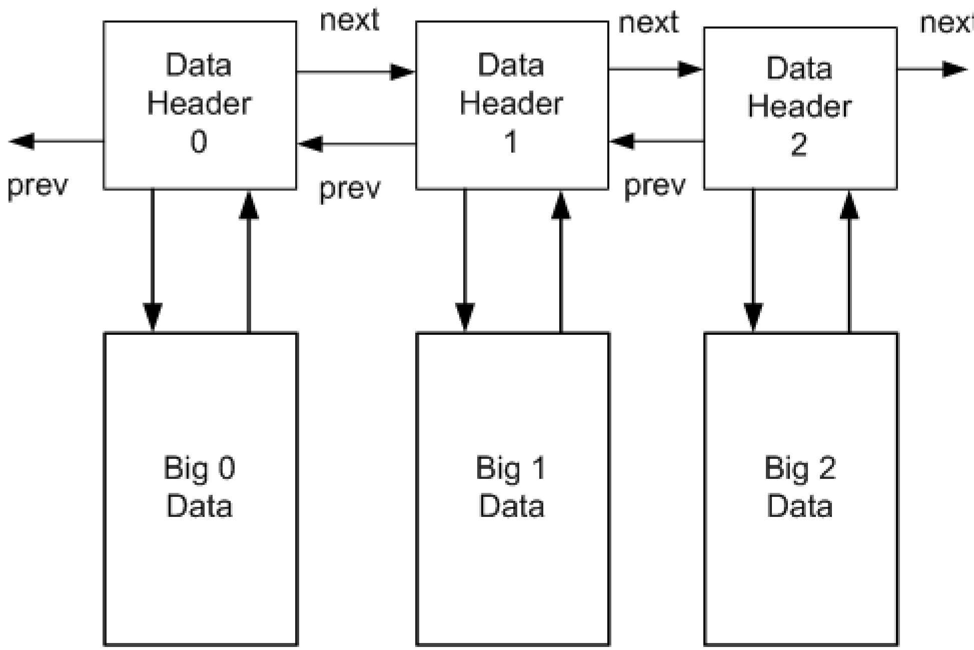
Converting data

- Writing converters
  - Convert the existing data to the new format.
    - Sometimes it's worth doing in real time
    - Generally it's better to do offline
- Dynamic memory
    - Now you need to track and manage more memory
    - Future optimizations
      - Use memory pools
- Single direction versus bidirectional
    - Prefer bidirectional,
      - Have a pointer that points from the cold side to the hot side
      - and vice versa
    - Can be single direction
      - Useful when you need that extra space of a pointer back.
      - Easier maintenance - Single links easier to maintain.
      - A pointer from the hot pointing to the cold

Take away

- Hot cold is a MACRO caching scheme
  - For those of you who are dyslexic (like me…)
    - It’s macro not marco!
  - It uses structures instead of individual data elements
- Can be scaled and reapplied all over the place.
  - Keep your eyes open
- The key to performance is caching
  - Data access is huge

New effective diagram



Thank You!

- Questions?