In [17]:
```
from notebook import *
# if you get something about NUMEXPR_MAX_THREADS being set incorrectly, d
```

Double Click to edit and enter your
1. Name
2. Student ID
3. @ucsd.edu email address

Lab 3: The Memory Hierarchy

The next two labs will explore the impact of memory accesses on program performance. By the end of them, you should have clearer understanding of the critical impact that a program's memory behavior has on its performance. In particular, you'll learn about the concepts of:

1. Memory alignment
2. Thinking in cache lines
3. Working sets
4. The cache hierarchy
5. The impact of miss rate on performance
6. The role of the TLB in determining performance
7. Spatial locality
8. Temporal locality
9. Cache-aware optimizations
10. The impact of data structures on memory behavior

Along the way, we'll address several of the "interesting questions" we identified in the first lab, including:

- Why does increasing the size of array change `CPI`? And why does this change occur so quickly?
- How and why do the datatypes we use change `IC` and `CPI`?
- Why does the order in which the program performs calculations affect `CPI`?

This lab includes a programming assignment.

Check the course schedule for due date(s).

1 FAQ and Updates

- There are no updates, yet.

2 Additional Reading

If, after these two labs, you still thirst for practical knowledge about using memory effectively, you should should read this series of articles: What every programmer should know about memory (https://lwn.net/Articles/250967/). It's long but quite good. It's not required reading, but, for the programming assignments that say "make this code go as fast you can," everything it includes is fair game.

3 Pre-Lab Reading Quiz

Part of this lab is a pre-lab quiz. The pre-lab quiz **has moved to Canvas** so I can allow multiple attempts. It is due **before class on the day the lab is assigned**. It's not hard, but it does require you to read over the lab before class. If you are having trouble accessing it, make sure you are **logged into Canvas**.

3.1 How To Read the Lab For the Reading Quiz

The goal of reading the lab before starting on it is to make sure you have a preview of:

1. What's involved in the lab.
2. The key concepts of the lab.
3. What you can expect from lab.
4. Any questions you might have.

These are the things we will ask about on the quiz. You *do not* need to study the lab in depth. You *do not* need to run the cells.

You should read these parts carefully:

- Paragraphs at the top of section/subsections
- The description of the programming assignment
- Any other large blocks of text
- The "About Labs in This Class" section (Lab 1 only)

You should skim these parts:

- The questions.

You can skip these parts:

- The "About Labs in This Class" section (Labs other than Lab 1)
- Commentary on the output of code cells (which is most of the lab)
- Parts of the lab that refer to things you can't see (like cell output)
- Solution to completeness questions.

3.2 Taking the Quiz

You can find it here: https://canvas.ucsd.edu/courses/29567/quizzes (https://canvas.ucsd.edu/courses/29567/quizzes)

The quiz is "open lab" -- you can search, re-read, etc. the lab.

You can take the quiz 3 times. Highest score counts.

4 Browser Compatibility [...]

5 About Labs In This Class [...]

6 Logging In To the Course Tools [...]

7 Grading

Your grade for this lab will be based on the following components

Part	value
Reading quiz	3%
Jupyter Notebook	45%
Programming Assignment	50%
Post-lab survey.	2%

No late work or extensions will be allowed.

We will grade 5 of the "completeness" problems. They are worth 3 points each. We will grade all of the "correctness" questions.

You'll follow the directions at the end of the lab to submit the lab write up and the programming assignment through gradescope.

Please check gradescope for exact due dates.

8 New Tools

8.1 Visualizing Memory Traces With Moneta

In this lab, we'll be using a tool called Moneta to visualize the memory accesses that a program makes. This will make it easier to see what a program is doing, and lets you develop your intuition for ideas like spatial locality, temporal locality, and working set size.

There are a couple of important aspect of Moneta that limit how many memory operations we can study at once.

1. The first is that Moneta is *trace based*. Tracing is a common technique for understanding computer performance. A *trace* is the general idea of a record of program behavior. For Moneta, the trace is a file that contains a list of all the memory accesses the program made and some information about them. Since modern processors can execute billions of instruction per second (many of which are memory accesses), traces can get very big, very fast. This means we will have to collect traces of just small portions of a program's execution -- otherwise, we'll fill up the file system on DSMLP.
2. To generate traces, Moneta uses a technique called *binary instrumention* which rewrites the executable to add some extra functionality. For Moneta, the extra functionality is writing each memory access to a file. Writing something to file can take thousands of instructions, so instrumenting a binary to produce a trace slows down execution *dramatically*. So, that's another reason we will only collect relatively small traces.
3. Finally, Moneta includes a graphical *trace viewer* (that runs in a Jupyter Notebook, conveniently enough). The viewer has to load the entire trace into memory. If the trace is too big, your Jupyter Notebook will crash, and your datahub session will abruptly stop working. This is likely to happen to you at least once. It is annoying. I apologize in advance. The way to avoid it is to use small traces.

Since instrumenting a binary to collect the trace slows it down so much, we cannot simultaneously measure a program's performance *and* collect a trace of its execution.

Enough about the implementation and limitations, let's see what Moneta can do.

Here's a fiddle with a simple program that takes an array and fills it with integers. There some extra stuff in here as well:

1. `TAG_START` and `TAG_STOP` are special functions that tell Moneta to label a set of memory accesses so we can look at them later.
2. `#include "pin_tags.h"` defines `TAG_START` and `TAG_STOP`. (If you're curious, it's at `/cse142L/CSE141pp-Tool-Moneta/moneta/pin_tags.h`. The implementations of the functions are interesting.)

`TAG_START` takes a tag name (`the-array`), a starting address (`data`), and an ending address (`&data[size]`). The final argument should be `false` for now. With this tag in place, the trace will record which accesses occurred between those two addresses and between the calls to `TAG_START` and `TAG_STOP`. We'll see why tagging is important in a moment.

Run the cell below, study the code, and answer the question. Then, we'll take a look at the trace.

In [ ]:
```
t = fiddle("array.cpp", function="array", run=["moneta"], opt="-O3", cmdl
code=r"""
#include"pin_tags.h"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* array(uint64_t * data, uint64_t size) {
    TAG_START("the-array", data, &data[size], false);
    for(unsigned int i = 0; i < size; i++) {
        data[i] = -i;
    }
    TAG_STOP("the-array");
    return data;
}

FUNCTION(one_array, array);
""")
compare([t.source, t.cfg])
```

Question 1 (Correctness - 3pts)

Based on the assembly and the source code and assuming `size = 1000` : How many instruction will execute in `array()` ? How many memory operations would you expect `array` to perform on `data` ? How many bytes of memory will be accessed? (Ignore instructions in functions that get called. Show your work.)

How many instructions?:

How many accesses?:

How many bytes accessed?:

In [ ]:
```
show_trace(t.mtrace, show_tag="the-array")
```

This is the Moneta interactive trace viewer. Primarily, it's a graph. The horizontal axis is "access number". Moneta numbers memory accesses as it records them. Program execution runs from left to right along the X axis which corresponds roughly to time.

The vertical axis is the *address offset*. Moneta's traces record the 64-bit address of every memory access, but those numbers are almost impossible to interpret. So, Moneta just plots the relative position of the accesses on the screen. The "Base address" value is the address that corresponds to 0 on the plot.

The viewer is interactive and the bar at the top provides several tools:

From left to right they are:

- Check boxes to lock the x or and y axes.
- Pan the image.
- Zoom to selected region
- Zoom in
- Zoom out
- Measure memory accesses in an area
- Reset zoom to show entire trace
- Go to previous zoom settings.
- Go to next zoom settings.
- Take a snapshot of the plot.

Question 2 (Completeness)

Based on the graph, how many memory accesses did `array` perform? How many bytes did it access?

How many accesses?:

How many bytes accessed?:

Show Solution

To see why we need tags, you can reset the zoom to show the entire trace. You'll see some horizontal lines at the very top and bottom of the graph. Roughly speaking, the call stack is at the top and grows down, while the heap (where `new` allocates memory) is at the bottom. Note the span of the vertical axis: My plot shows a range of 4.5×10^{13} bytes or about 40TB. The address space is huge! This is why we need tags. It's really hard to find anything otherwise.

Interesting Question: How can the program access a range of memory addresses that is vastly larger than the amount of memory on the machine?

Play around with the viewer a bit, but there are more interesting traces to come.

8.2 Measuring Cache Performance with Performance

Counters

We'll use performance counters to measure cache performance on our bare metal machines in the cloud. We'll use the same tools we've used so far to measure `IC`, `CPI`, and `CT`.

Let's measure the cache behavior of `array` from the example above. Here's the same example again:

In [ ]:
```
t = fiddle("array.cpp", function="array", run=["perf_count"], opt="-O3", 
code=r"""
#include"pin_tags.h"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* array(uint64_t * data, uint64_t size) {
    TAG_START("the-array", data, &data[size], false);
    for(unsigned int i = 0; i < size; i++) {
        data[i] = -i;
    }
    TAG_STOP("the-array");
    return data;
}

FUNCTION(one_array, array);
""")
compare([t.source, t.cfg])
```

Question 3 (Completeness)

If `size == 8192`, how many total cache misses do think will occur during the execution of `array()`? (assume the cache is empty to begin and that cache lines are 64 bytes).

Misses for size 8192:

Average misses per instruction in this loop:

Show Solution

8.3 Tensors

In this lab and future labs, we'll be using a data structure called a *tensor*. Tensors are a hot topic nowadays (you may have heard of the TensorFlow programming system from Google that accompanies their Tensor Processing Unit specialized processor), but at their heart they are not complicated: They are just multi-dimensional arrays.

We'll be using a 4-dimensional tensor data type called `tensor_t`. It's defined in `/cse142L/CSE141pp-SimpleCNN/CNN/tensor_t.hpp`. Here's the key parts of the source code:

```
template<typename T>
struct tensor_t
{

    tdsize size;
    T * data;

    tensor_t( int _x, int _y, int _z, int _b ) :  size(_x, _y, _z,
_b) {
        data = new T[size.x * size.y * size.z * size.b]();
    }

      T& get( int _x, int _y, int _z, int _b=0 ) {
        return data[
            _b * (size.x * size.y * size.z) +
            _z * (size.x * size.y) +
            _y * (size.x) +
            _x
            ];
    }
};
```

Note that `tensor_t` is a template, so we can have tensors of many different types: `tensor_t<int>`, `tensor_t<float>`, etc.

Although `tensor_t` is 4-dimensional, it stores its contents in a one-dimensional array of `T` called `data`. The `get()` method maps coordinates into that linear array and returns a reference to the corresponding entry. Since `get()` returns a reference, we can say things like:

```
tensor_t<float> t(10, 10, 10, 10);
t.get(1,2,3,4) = 4.0;
float s = t.get(1,2,3,4);
```

Question 4 (Correctness - 2pts)

Given the following `tensor_t` declarations and accesses, compute the total size of the tensor in element and bytes, and the index (starting at 0) of the corresponding data elements in the `data` array.

1. `tensor_t<uint32_t>(3,5,6,7)`
   A. Total elements:
   B. Total bytes:
   C. `get(1,0,0,0)`:
   D. `get(0,1,0,0)`:
   E. `get(0,0,1,0)`:
   F. `get(0,0,0,1)`:
2. `tensor_t<double>(2,4,8,16)`
   A. Total elements:
   B. Total bytes:
   C. `get(1,3,2,4)`:
   D. `get(2,2,1,7)`:

Here's a simple example of using a `tensor_t<uint32_t>` as a 2-dimensional array. The `z` and `b` dimensions have a size of 1, so they act as if they don't exist.

In [ ]:
```
t = fiddle("tensor.cpp", function="tensor", run=["moneta"], tagged_only=T:
code=r"""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* tensor(uint64_t * data, uint64_t size) {

    tensor_t<uint32_t> t(4,8,1,1);
    TAG_START("init", t.data, &t.as_vector(t.element_count()), false);

    for(uint y = 0; y < 8; y++) {
        for(uint x = 0; x < 4; x++) {
            t.get(x,y,0,0) = x;
        }
    }

    TAG_STOP("init");
    return data;
}

FUNCTION(one_array, tensor);
""")
compare([t.source, t.cfg])
show_trace(t.mtrace, show_tag="init")
```

The graph is zoomed *way* in so you can see individual accesses and memory locations. If you look carefully, you can see that each element of the tensor is 4-bytes. You can also see that, as we

iterate through the `x` and `y` dimensions, the memory accesses move sequentially through a continuous region of memory. If you look carefully at assembly for the loop, you'll see a nice example of function inlining and constant propagation.

8.4 The Miss-Machine

We're going to need to generate some cache misses very efficiently in this lab. It's actually pretty hard to reliably create cache misses efficiently, especially if you want to out-smart modern cache hardware mechanisms like prefetchers. We also need to precisely control the amount of memory it touches. You could use a random number generator, but even the simplest ones are pretty computationally intensive. Instead, we are going to use a clever data structure that I don't think has an official name, so I'm going call it the "miss machine".

The miss-machine is a circular linked list. The links are allocated in a block (corresponding to the memory footprint we want) and then formed into a circular linked list. Then, you set the `next` pointer so that the list jumps around at random, hitting every link, and eventually returning to the start. Then, if you want cache misses, you just traverse the list for as long as you like. The sequence of addresses is (as near as the cache can tell) random, and the code required is extremely small -- just a few instruction are sufficient.

Here's an implementation. Read through the code and comments carefully.

In [ ]:

```
t = fiddle("miss_machine.cpp", function="miss_machine", opt="-O1 ", run=[
code=r"""
#include<cstdint>
#include<vector>
#include<algorithm>
#include"function_map.hpp"

struct MM {
    struct MM* next;  // I know that pointers are 8 bytes on this machine
    uint64_t junk[7]; // This forces the struct MM to take a up a whole c
};

extern "C"
struct MM * miss(struct MM * start, uint64_t count) {
    for(uint64_t i = 0; i < count; i++) { // Here's the loop that does th
        start = start->next;
    }
    return start;
}


extern "C"
uint64_t* miss_machine(uint64_t * data, uint64_t size, uint64_t arg1) {
#define ARRAY_SIZE (8*1024)
    auto array = new struct MM[8*1024]; // This is bigger than the L1

    // This is clever part  'index' is going to determine where the point
    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < ARRAY_SIZE; i++) {
        index.push_back(i);
    }
    // Randomize the list of indexes.
    std::random_shuffle(index.begin(), index.end());

    // Convert the indexes into pointers.
    for(uint64_t i = 0; i < ARRAY_SIZE; i++) {
        array[index[i]].next = &array[index[(i + 1) % ARRAY_SIZE]];
    }

    MM * start = &array[0];
    start = miss(start, 1024*1024*128); // 128 million accesses.
    return reinterpret_cast<uint64_t*>(start); // This is a garbage value
}

FUNCTION(one_array_1arg, miss_machine);
""", cmdline=f"--size 1024", perf_cmdline="--stat-set L1.cfg --MHz 3500")
display(t.source)
f = fiddle("miss_machine.cpp", function="miss", opt="-O1")
```

Here's the assembly for `miss()`

In [ ]:
```
display(f.cfg)
```

Question 5 (Completeness)

Based on this code, how many misses per instruction would you expect this code to produce? We'll call this metric `L1_MPI`. It's just the number of L1 cache misses divided by `IC`.

`L1_MPI`:

Show Solution

Here's the results:

In [ ]:
```
display(render_csv('miss_machine.csv', columns=["IC", "L1_cache_misses",
```

A few things are notable: `L1_MPI` and `CPI` are the largest numbers we have seen to date. The miss machine works (But it could work better).

Question 6 (Optional)

Modify the fiddle above to get `L1_MPI` above 75%. Think about how you would modify the assembly to increase `L1_MPI`. How can you make that happen *without* manually editing the assembly?

9 Thinking in Cache Lines

One of the central issues in cache-aware programming (i.e., crafting your code to make efficient use of the memory hierarchy) is understanding "how much" memory your program (or part of it) is using and "how big" a particular data structure (e.g., an array) is, and how often your program accesses memory. It turns out there are a surprising number of ways to measure these things and the answers can be unexpectedly surprising and non-intuitive. Not only that, the most obvious ways to measure these characteristics are not the most useful if you're interested in fully-utilizing your memory system.

So let's take a look at three questions and see how to answer them in a cache-aware way.

9.1 How big is my data structure?

9.1.1 Primitive Types

Let's start simple: How many bytes do each of the primitive C++ data types occupy? Here's some code to check. It uses C's `sizeof` operator (it's not technically a function. Why?) that tells you how many bytes something occupies. The types listed are all the C++ primitive types.

In [ ]:
```
t = fiddle("primitive_sizes.cpp", function="primitive_sizes", run=["local
code=r"""
#include<cstdint>
#include<iostream>
#include"function_map.hpp"

extern "C"
uint64_t* primitive_sizes(uint64_t * data, uint64_t size, uint64_t arg1)
    std::cout << "\n";
    std::cout << "sizeof(char) = " << sizeof(char) << "\n";
    std::cout << "sizeof(short int) = " << sizeof(short int) << "\n";
    std::cout << "sizeof(int) = " << sizeof(int) << "\n";
    std::cout << "sizeof(long int) = " << sizeof(long int) << "\n";
    std::cout << "sizeof(long long int) = " << sizeof(long long int) << "
    std::cout << "sizeof(float) = " << sizeof(float) << "\n";
    std::cout << "sizeof(double) = " << sizeof(double) << "\n";
    std::cout << "sizeof(long double) = " << sizeof(long double) << "\n";
    std::cout << "sizeof(int8_t) = " << sizeof(int8_t) << "\n";
    std::cout << "sizeof(int16_t) = " << sizeof(int16_t) << "\n";
    std::cout << "sizeof(int32_t) = " << sizeof(uint32_t) << "\n";
    std::cout << "sizeof(int64_t) = " << sizeof(int64_t) << "\n";
    std::cout << "sizeof(int64_t*) = " << sizeof(int64_t*) << "\n";
    std::cout << "sizeof(void*) = " << sizeof(void*) << "\n";
    return data;
}

FUNCTION(one_array_1arg, primitive_sizes);
""")
```

The types with a number in their names (like `uint64_t`) let you specify specific sizes (in bits) you'd like to use. They are defined in the `cstdint` header. The C/C++ standard (https://en.wikipedia.org/wiki/C_data_types) doesn't give specific sizes for 'int', 'long int', etc. Instead, it places some constraints on what possible values might be. This is a design "bug" in the languages. I don't know of any other modern languages that leave the bit widths of primitive types unspecified. One reason is that it hurts portability. The other, is that it makes cache-aware programming harder (as we shall see).

9.1.2 Structs

Let's try something more complicated. Look at the code below and answer this question. Then, run the code:

Question 7 (Completeness)

Predict how many bytes of the structs below will occupy.

struct	sizeof()
`struct_1`	
`struct_2`	
`struct_3`	
`struct_4`	
`struct_5`	
`struct_6`	
`struct_7`	
`struct_8`	

In [ ]:

```
t = fiddle("struct1.cpp", function="struct_size", run=["local"],
code=r"""
#include<cstdint>
#include<iostream>
#include"function_map.hpp"

struct struct_1 {
    uint32_t a;
};

struct struct_2 {
    uint32_t a;
    uint32_t b;
};

struct struct_3 {
    uint32_t a;
    uint8_t b;
};

struct struct_4 {
    uint64_t a;
    uint8_t b;
};

struct struct_5 {
    uint8_t a;
    uint8_t b;
};

struct struct_6 {
    uint64_t a;
    uint8_t b;
} ;

struct struct_7 {
    uint64_t a;
    uint8_t b;
    uint8_t c;
} ;

struct struct_8 {
    uint8_t a;
    uint64_t b;
    uint8_t c;
} ;


extern "C"
uint64_t* struct_size(uint64_t * data, uint64_t size, uint64_t arg1) {
    std::cout << "\n";
    std::cout << "sizeof(struct_1) = " << sizeof(struct_1) << "\n";
    std::cout << "sizeof(struct_2) = " << sizeof(struct_2) << "\n";
    std::cout << "sizeof(struct_3) = " << sizeof(struct_3) << "\n";
    std::cout << "sizeof(struct_4) = " << sizeof(struct_4) << "\n";
    std::cout << "sizeof(struct_5) = " << sizeof(struct_5) << "\n";
```

```
    std::cout << "sizeof(struct_6) = " << sizeof(struct_6) << "\n";
    std::cout << "sizeof(struct_7) = " << sizeof(struct_7) << "\n";
    std::cout << "sizeof(struct_8) = " << sizeof(struct_8) << "\n";
    return data;
}

FUNCTION(one_array_1arg, struct_size);
""")
```

```
![mind_blown.gif](attachment:mind_blown.gif)
```

What's going on?!?!

What's going on is *data alignment*. If a memory address, A, is n-byte aligned, then $A \mod n = 0$. A particular value is "width-aligned" if it is aligned to its own size in bytes. So, a width-aligned `uint64_t` would reside at an address that is a multiple of 8 bytes.

In most architecture, width-aligned access is faster than non-width-aligned access. In some architectures, the ISA does not directly support non-width-aligned access (i.e., you can't load a 64-bit value from an address that is not a multiple of 8), because it requires extra hardware and complexity (e.g., if the architecture allows unaligned, multi-byte values, then a single load can access *two* cache lines instead of one). Instead, they require the compiler to implement these accesses with loads and shifts.

The strangeness in the outputs of the `sizeof` is a product of this. The C/C++ standards require that the members of a struct be stored *in the order they are declared*. Modern compilers "pad" members of the struct to enforce width-alginment. For this purpose, it's assumed that the struct starts address 0.

For example, consider `struct_8` above. It's laid out like so:

Byte	
0	a
1	unused
2	unused
3	unused
4	unused
5	unused
6	unused
7	unused
8	b
9	b
10	b
11	b
12	b
13	b
14	b
15	b
16	c
17	unused

Byte	
18	unused
19	unused
20	unused
21	unused
22	unused
23	unused

If this seems inefficient... it is. Or at least, it's a trade-off. It's better for performance this way and memory is plentiful. Really, though, the programmer should re-order the fields of the struct to allow for a more efficient layout. See if you can re-arrange the fields in `struct_8` to make it fit in 16 bytes.

9.1.3 Arrays

Compared to structs, arrays are pretty well-behaved. The size of an array is just the number of elements in the array multiplied by the size of the struct:

In [ ]:
```
t = fiddle("struct1.cpp", function="struct_size", run=["local"],
code=r"""
#include<cstdint>
#include<iostream>
#include"function_map.hpp"

struct struct_8 {
    uint8_t b;
    uint64_t a;
    uint8_t c;
} ;


extern "C"
uint64_t* struct_size(uint64_t * data, uint64_t size, uint64_t arg1) {
    struct struct_8 _8[3];
    std::cout << "\n";
    std::cout << "sizeof(struct_8[3]) = " << sizeof(_8) << "\n";
    return data;
}

FUNCTION(one_array_1arg, struct_size);
""")
```

9.2 How much memory does my code access?

Above, we measured the size of a data structure or array in bytes, and this makes sense for thinking about its size.

A related question is how much data does my program access. If you are interested in writing code that is cache-aware, thinking of data measured in bytes is not that useful. A better choice is to think about data measured in cache lines, because cache lines are the units of memory that the memory hierarchy transfers between caches.

So what is one cache line of memory? The seemingly obvious answer is that is the number of bytes that a cache line holds. If that were the case, we could just divide the size of a structure by the cache line size. But there is more to it: Cache lines are width-aligned. That means that each cache line of memory starts at an address that is divisible by the cache line's size. This means that the number of cache lines a struct occupies *depends on its alignment*.

For example, let's assume our cache line size is 16 bytes, and `my_struct` is 16 bytes long. Here are two possible scenarios for `my_struct`. In the first, the beginning of `my_struct` is aligned to a cache line boundary, and `my_struct` occupies 1 cache line.

byte	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
cache line	0																1											
	my_struct																											

In the second scenario, `my_struct` is not aligned, and it occupies two cache lines:

byte	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
cache line	0																1											
			my_struct																									

This means that if we access the whole struct, we will incur one compulsory cache miss. But in the second situation we will have two.

We can modify the miss machine to illustrate this situation. The code is below. There are three differences:

1. I changed the name of the fields in `MM`. That's just to make the table below more readable.
2. `miss()` reads from two fields of `MM`. `a` at the beginning and `h` at the end.
3. We allocate `array` in a complicated way.

For `array`, instead of using `new`, we use `posix_memalign()` (https://man7.org/linux/man-pages/man3/posix_memalign.3.html) which lets you set the alignment of the allocated memory with its secord argument. We allocate `array` so that the elements it contains will be width-aligned. That is, `array % sizeof(MM) == 0`.

Having carefully, allocated aligned memory, we then unalign it:

```
array = reinterpret_cast<A*>(reinterpret_cast<uint64_t*>(array) + a
rg1);
```

The line above shifts `array` by `arg1` 8-byte words. So, if `arg1 = 4`, then `array % sizeof(MM) == 32`. You can read about `reinterpret_cast` (https://en.cppreference.com/w/cpp/language/reinterpret_cast), but you should use it sparingly (They gave it a hard-to-type name on purpose. Really!).

So, if we call this function with multiple values of `arg1`, we'll have something like this:

8-byte word	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
64-bytes cache line	00								01								02											
arg1 == 0	a							array[0]	h	a						array[1]	h	a						array[2]	h	a		
arg1 == 1		a						array[0]	h	a						array[1]	h	a						array[2]	h	a		
arg1 == 2			a					array[0]	h	a						array[1]	h	a						array[2]	h	a		
arg1 == 3				a				array[0]	h	a						array[1]	h	a						array[2]	h	a		
arg1 == 4					a			array[0]	h	a						array[1]	h	a						array[2]	h			
arg1 == 5						a		array[0]	h	a						array[1]	h	a						array[2]				
arg1 == 6							a	array[0]	h	a						array[1]	h	a						array				
arg1 == 7								a	array[0]	h	a					array[1]	h	a						ar				

When `arg1 % 8 = 0`, a `MM` occupies one cache line. When `arg1 % 8` is anything else, it occupies 2. And since `miss()` accesses `h`, we will access both lines.

Let's run the code with a range of `arg1` values.

In [ ]:

```
t = fiddle("unaligned.cpp", function="miss_machine", opt="-O1", run=['per
code=r"""
#include<cstdint>
#include<iostream>
#include<vector>
#include<algorithm>
#include"function_map.hpp"
#include"archlab.hpp"

struct MM {
    struct MM* a;
    uint64_t b;
    uint64_t c;
    uint64_t d;
    uint64_t e;
    uint64_t f;
    uint64_t g;
    uint64_t h;
};

extern "C"
struct MM * miss(struct MM * start, uint64_t count) {
    uint64_t sum = 1;
    for(uint64_t i = 0; i < count; i++) { // Here's the loop that does th
            sum += start->h;
            start = start->a;
    }
    return start + sum;
}

extern "C"
uint64_t* miss_machine(uint64_t * data, uint64_t size, uint64_t arg1) {

#define ARRAY_SIZE ((32*1024)/sizeof(struct MM)* 4*8)
#define ITERATIONS 1024*1024*128
    struct MM * array = NULL;
    int r = posix_memalign(reinterpret_cast<void**>(&array), sizeof(struc
    assert(r == 0);
    array = reinterpret_cast<MM*>(reinterpret_cast<uint64_t*>(array) + ar

    std::vector<uint64_t> index;
    for(uint64_t i = 0; i < ARRAY_SIZE; i++) {
        index.push_back(i);
    }
    std::random_shuffle(index.begin(), index.end());

    for(uint64_t i = 0; i < ARRAY_SIZE; i++) {
        array[index[i]].a = &array[index[(i + 1) % ARRAY_SIZE]];
    }

    std::cout << "alignment of array" << ((uint64_t)array % sizeof(struct
    MM * start = &array[0];
    start = miss(start, ITERATIONS);
    return reinterpret_cast<uint64_t*>(start);
}
```

```
FUNCTION(one_array_1arg, miss_machine);
""", cmdline=f"--size {1024} --arg1 0 1 2 3 4 5 6 7", perf_cmdline="--sta

f = fiddle("unaligned.cpp", function="miss", opt="-O1")
```

Here's the assembly for `miss()`:

In [ ]:

```
display(f.cfg)
```

Question 8 (Completeness)

Based on the exercises about the miss machine and the code above, predict the `L1_MPI` for `miss()` when `array` is aligned (`arg1= 0`) and unaligned (`arg1 != 0`).

aligned `L1_MPI`:

unaligned `L1_MPI`:

Show Solution

9.3 How often does my code access memory?

Thinking in terms of cache lines doesn't stop with *how much* memory a program accesses, it can also extend to *how often* your program accesses memory. In highly-optimized systems, it often makes more sense to think about how many cache misses your code incurs rather than how many memory accesses it makes.

You might have noticed that in CSE142 the lecture focuses on "cache miss rate" which is a good measure of how effectively the cache can exploit locality. In this lab, however, we measure misses per instruction (MPI). The distinction is important: If we have few memory accesses, a high miss rate is not necessarily bad. And if we have many, many memory accesses even a low miss rate could have a large impact on CPI (and, therefore, execution time.

Here's another example I think is illustrative. It comes from a research project called RamCloud (https://ramcloud.atlassian.net/wiki/spaces/RAM/overview). RamCloud was a DRAM-based storage system that was highly optimized for low latency. Here's a figure from one of their presentations

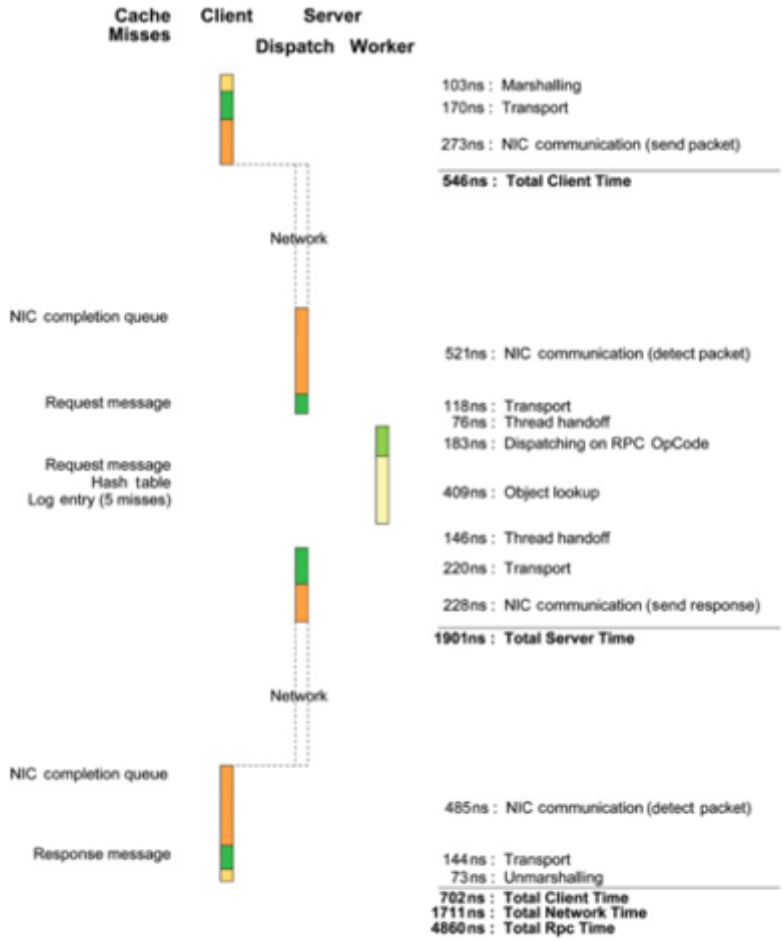


Fig. 10. Timeline to read a 100-byte object with 30-byte key chosen at random from a large table; vertical distance represents time to scale. There were a total of nine cache misses on the server and two on the client; the text in the left column identifies the cause and approximate time of occurrence for each miss. The total network time (including both request and response) was 1,711ns (the experiment could not measure request and response times separately).

The figure is a very careful analysis of the execution of a RamCloud client talking to a RamCloud server. You don't need to understand the figure in detail, but what's instructive is that and the main focus is the number of cache misses incurred rather than the number of instructions, the number of memory operations, or the amount of data accessed. Focusing on cache misses also means, implicitly, that this analysis is done in terms of cache lines rather than byte or words.

9.4 Language Support

The experiments above show that alignment and struct layout can affect the performance of toy programs, but do these kinds of details really make any difference in real code? They do!

How can we tell? Two ways:

1. We could go and write some high-performance systems and watch as alignment-related performance problems appear.
2. We can look at the tools that languages provides to deal with these issues.

#1 is lots of fun and I recommend it, but it takes a long time and we only have one quarter. But there are three programming assignments left...

Let's look at #2. For a long time, compiler writers and language designers have realized that memory layout and alignment is important, so they have provided support in the language to help programmers deal with this.

9.4.1 Struct Initialization In C

Since the early versions of C, you can initialized a struct like this:

```
struct Foo {
    char a;
    int b;
    char c;
};

Foo foo = {7,4,3};
```

Which will set `foo.a = 7`, `foo.b = 4`, and `foo.c = 3`. We'll call this "positional initialization" or PI. I think PI is super-unreadable, but it has an even bigger problem: If you change the layout of `Foo` to improve it's cache efficiency, you have to update every static initialization of a `Foo` and there's no way for the compiler to tell if you missed one. So the C99 standard gave us the "designated initialized":

```
Foo foo = {.a = 7, .b=4, .c=3};
```

This features appears as a gcc extension before it showed up in the C standard, and it's main use case was making easier to adjust struct layouts for memory-efficiency reasons. For instance, in the unaligned access example above, we could move `MM.h` to be near `MM.a` and reduce the effects of poor alignment.

9.4.2 C++ Object Alignment

Usually, the compiler can do a good job of aligning your structs, but there are cases where the programmer needs to force a structure to be aligned to a particular size. The old fashioned way to do this was by wrapping `malloc()` (the C version of `new`) to create a new memory allocation function that could return memory at the desired alignment. Writing such a function is a moderately interesting programming exercise.

Since 2001, C has had support for aligned allocation through `posix_memalign()`.

C++ has a more elegant solution and allows you to annotate the type with its alignment requirement using `alignas()`:

```
struct alignas(32) Foo {
    char a;
    int b;
    char c;
};
```

Which guarantees that all instance of `Foo` will be so aligned to 32 bytes.

For both `posix_memalign()` and `alignas()`, the alignment value must be a power of 2. Other alignments are not generally useful.

10 Latency and Bandwidth

Latency and bandwidth are two fundamental measures of memory performance.

- Latency -- in seconds (or sometime cycles) -- measures the time it takes for a single memory access to complete.
- Bandwidth -- measured in bytes, megabytes, or gigabytes per second -- is the amount of data the processor can access per unit time.

Caches can improve both latency and bandwidth: The L1 cache has lower latency and higher bandwidth than the L2. The L2 out-performs the L3 on both metrics, and the L3 is better than DRAM on both metrics.

Let's start by taking some baseline measurements of the the bandwidth and latency capabilities of our test machine. We'll start with DRAM.

10.1 Measuring DRAM Latency and Bandwidth

Before we measure DRAM's performance, we should be precise about what exactly we mean by "latency" and "bandwidth" for DRAM.

For DRAM latency, we mean the number of seconds it takes for a load instruction to retrieve data assuming that the load misses in *all* levels of the cache. This means that the memory request "goes all the way to memory" and has to come all the way back.

For DRAM bandwidth, we mean the maximum number of bytes the processor can read or write per second from DRAM rather than from any of the caches. This means that that the only bytes that count for DRAM bandwidth are those read or written as part of a cache miss.

Measuring these two values is surprisingly difficult and would make a very good (and pretty challenging) programming assignment for a graduate-level architecture course. It's made more difficult in modern machines by the presence of multiple processors, multiple DRAM interfaces, multiple caches, and multiple cores. We'll discuss some of this complexity in more detail when in future labs when we address the impacts of parallelism, but for now we will stick the simplest version of these questions.

Given the difficulty of measuring these values, we will rely on Intel's Memory Latency Checker (https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html). It can perform a huge array of measurements, but we'll just do two:

In [ ]:
```
!cse142 job run --lab caches 'mlc --bandwidth_matrix; mlc --latency_matri
```

The output is a little verbose, but you should see two numbers that look like measured values rather than documentation. They are both for 'Numa node' 0. The first number is the bandwidth in

MB/s. The second is latency.

Question 9 (Completeness)

Fill in the table below with the values you measured.

metric	value
Latency (ns)	
Latency @ max clock rate (cycles, for our processor)	
Latency @ min clock rate (cycles, for our processor)	
Bandwidth (GB/s)	

Show Solution

Let's see what kind of hardware is providing this kind of performance. To do this we'll use `dmidecode`, a utility to read the systems DMI table. The DMI table is populated by the system's BIOS (the software runs immediately after you turn the computer on) to describe the hardware available. DMI stands for "Desktop management interface" which is may be the least descriptive name for anything we'll use in the course. Everyone just calls it the "DMI table".

Anyway, `dmidecode` will dump all sort of interesting information about the system, but we'll pass `-t memory` to just ask it about the DIMM memory modules installed. We'll also use grep to filter out some extraneous information:

In [ ]:
```
!cse142 job run --lab caches 'dmidecode -t memory > dmi_output.txt'
!grep -v 'Not Specified\|Unknown\|None' dmi_output.txt
```

There's a lot of information here, but let's walk through it: Each entry in a DMI table has a type and a unique ID number or "Handle". Type 0x17 is a memory module (or at least a slot for a memory module). Type 0x11 is a "memory array" (for me it shows up as Handle 0x0011). Our system has one memory array with 4 DIMM slots (Handles 0x0020-0x0023), but only one of them (0x0021 for me) is populated.

We can see that this module is 16GB in side, is manufactured by Micron, runs at a supply voltage of 1.2V, is DDR4, and runs at 2667MT/s.

MT/s stands for mega-transfers per second and measures how often data flows across the 64-bit memory bus.

Question 10 (Completeness)

What is the peak memory bandwidth of our system? What fraction of the peak bandwidth was `mlc` able to attain?

metric	value
Peak bandwidth	
`mlc` % of peak	

Show Solution

10.1.1 Is DRAM Fast?

60ns is an almost unimaginably short period of time, and 18GB/s seems like a lot of data, but we need to think about this relative to the processor. If you look back at the the `CPI` measurements you collect for Lab 1, you'll probably find CPI values as low as 0.5. At 3.6GHz, that means our processor is executing around 7 billion instructions per second or, on average, one instruction every 0.166ns. So 60ns is about 360 instructions.

For bandwidth, the situation is not great either. On average about 20% of instruction access memory and every instruction must be loaded from memory to execute. The average length of an x86 instruction is about 2 bytes and for simplicity, let's assume all memory accesses are 4 bytes. That means that, on average, each instruction needs `2 + 0.2 * 4 = 2.8` bytes of memory.

Let's compare that to what our measurements show: 18GB/s divided by 7 Billion instructions per second is about 2.6 bytes per instruction. Not far off! However, recall from lab one that our processor actually 6 cores! And the memory bandwidth from it's single memory channel will be shared among all of them.

We should also keep in mind that our machines are memory-poor: Campus bought them to be as cheap and compact as possible. So they are physically small (so they only have 4 memory slots) and campus only ordered one DIMM for each.

We will need a way to get higher bandwidth and lower latency.

10.2 Measuring Cache Latency and Bandwidth

Caches reduce latency and increase bandwidth. Let's see by how much:

10.2.1 Cache Latency

We will use a different tool to measure cache latency. This one is called `lat_read_rd` and it's part of set of system benchmarks called `lmbench` (http://lmbench.sourceforge.net/). `lat_mem_rd` measures the latency for repetitive accesses large and larger arrays. For arrays that fit in the L1 cache, the average latency of these accesses is the L1 cache latency. For arrays bigger than the L1 but that fit in the L2, it's the L2 cache latency. Likewise, for L3.

Question 11 (Optional)

How would you implement `lat_mem_rd` ?

Show Solution

Recall from Lab 1 that our processor has a 32KB L1, a 256KB L2, and a 12MB L3. Usually, I'd have you run `lat_mem_rd` but it takes a long time to run and the output format is annoying, so here are the results:

In [ ]:
```
df = render_csv("lat_mem_rd.csv")
df["size_bytes"] = df["size_MB"]*1024*1024
plotPE(df=df, what=[("size_bytes", "latency_ns")], lines=True, logx=2, lo
```

The "stair step" pattern you see is the jump in latency as data accesses start being served out of the slower and slower, layers of the memory hierarchy. The steps occur roughly we expect them to: 32kB (L1 to L2), 256kB (L2 to L3), and 12MB (L3 to DRAM). For larger sizes things get noisier so the "steps" aren't as crisp.

Here's the raw data:

In [ ]:
```
display(df[["size_bytes", "size_MB", "latency_ns"]])
```

Question 12 (Correctness - 4pts)

Based on this graph, give the latencies for each of the caches.

Level of the memory hierarchy	latency (ns)	latency (Cycles)
L1		
L2		
L3		
Main memory		

Happily, both `mlc` and `lat_mem_rd` agree on the main memory latency.

We can also check how our measurements compare to the processor specifications. For the L1 cache, Intel has disclosed that the minimum latency for the L1 read is 4 cycles. That should match the value you computed above.

the value you computed above.

As far as I know, Intel has not release a similar value for the L2 and L3.

10.2.2 Cache Bandwidth

We don't have a handy tool for measuring cache bandwidth. Maybe we'll build one as a future programming assignment...

We do however, have some technical details about about processor, so we can calculate what the peak (maximum attainable) cache bandwidth should be.

Skylake processors (of which our CPU is an example) can execute 2 64-byte loads and 1 64-byte store per cycle. At 3.5GHz, this works out to `2*64*3.5e9 = 417GB/s` for loads and `213GB/s` for stores for a total 625GB/s.

The L2 cache provide one 64-bytes load *or* store per cycle for `213GB/s.`

I don't have any information about the L3.

10.2.3 Are the Caches Fast?

They are certainly better than DRAM -- the L1's latency is `61/1.147 = 52.3` times lower than DRAM, and bandwidth (in our system with one bank of DRAM) is `625/18.8 = 33` times higher. Not only that, remember that we have six cores, so the total L1 cache bandwidth is `625GB/s * 6 = 3.7TB/s .`

However, the minimum load latency -- 4 cycles -- is still pretty high, considering that it's 4 times longer than the latency for a simple integer arithmetic operation (add, sub, etc.). That means that having lots of load instructions can have a significant impact on CPI -- remember that unoptimized code from Lab 2 with all accesses to local variables on the stack? The 4 cycle L1 latency is part of why `-O0` is so slow.

Caches are also slow compared to the register file. Skylake's register files have a total bandwidth (at 3.5GHz) of something like 5TB/s (although it's very hard to fully utilize it.) per core (or 30TB/s across all 6 cores). The register file latency is a little hard to quantify in a way that is comparable to cache latency, but 1/2 a cycle is a reasonable approximation.

The register file is also quite large: The total size is about 1kB. It's reasonable to think of the register file a software-managed, "L0" cache.

11 Locality In Space and Time

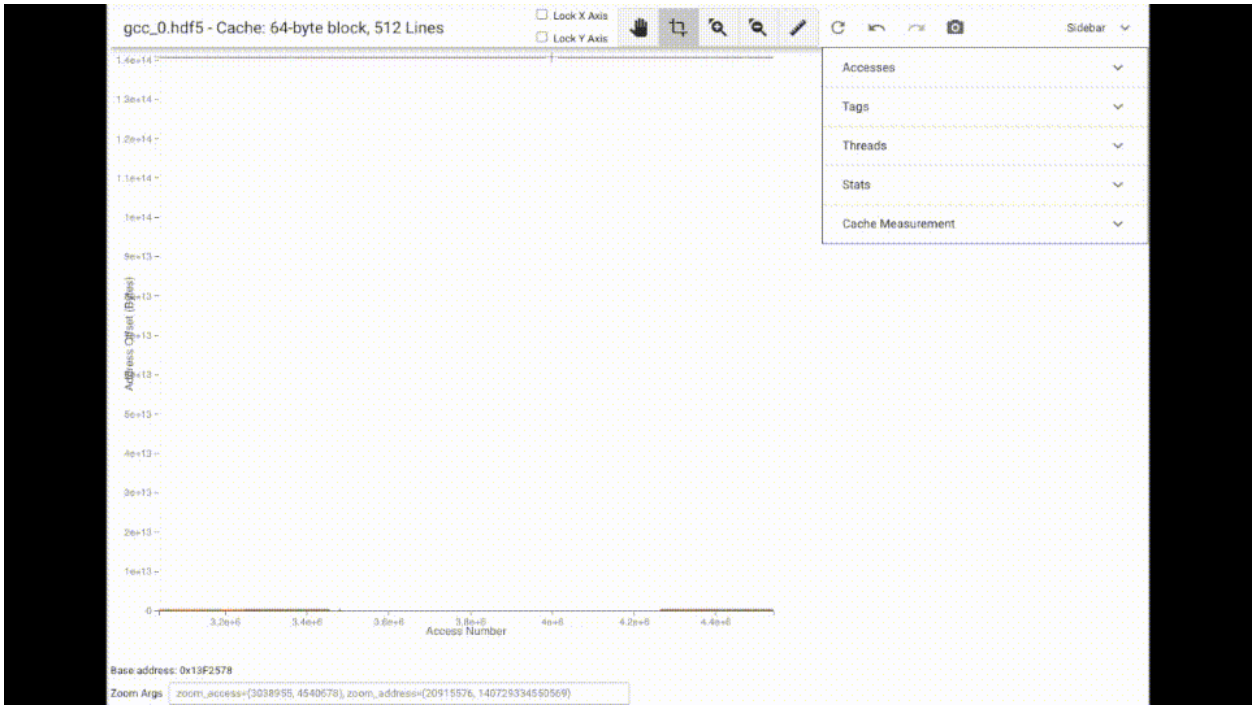
Caches improve program performance by exploiting the fact that memory accesses are not random: Programs access memory in patterns and there is a lot of repetition in what programs do and a lot of similarity how programs behave at different times. This is not surprising since, programs are written languages with constructs like loops and function calls: These constructs naturally give rise to patterns because they cause the same code to execute repeatedly.

Let's take a look at a real program to see what this regularity looks like in real life. The command below will collects a trace for the first 10 million memory accesses as `g++` starts compiling a complex program. (If you're curious, the program is `cc1plus`, the core of `g++`).

There are no tags in the trace, so finding things is hard. Here's what todo:

1. Freeze the x-axis (you'll just zoom in and out vertically)
2. Select the zoom-to-region tool.
3. Select one of the horizontal lines that visible in the initial view of the trace.
4. Repeatedl zoom in on one of the visible horizontal lines. Do this between 4 and 7 times. Eventually details will emerge.
5. Reset the zoom and look at a different line.

For example:



In [ ]:

```
!mtrace --trace gcc --memops 10000000 --skip 10000000 -- /usr/lib/gcc/x86_
show_trace("./gcc_0.hdf5")
```

```
Go exploring a little bit.  Places to look:

1. The stack:  Keep zooming in on the top-most horizontal line in the
plot.
2. The Heap: Zoom in on the bottom-most line until it resolves into to
several lines.  Then, zoom in on the top one.

Things to look for:

1. Areas with repeated patterns of memory access.
2. Memory locations that are accessed frequently
3. Areas (in space and time) where accesses appear random.
4. Periods of sequential access.
```

```
If you find something interesting, click the camera icon to grab a screen
capture:

![image.png](attachment:image.png)

Let's look at locality in a more controlled environment.
```

11.1 Spatial Locality

As you should recall from lecture, *spatial locality* is a property of a program (or part of a program) where it accesses memory locations nearby locations it has accessed recently. Let's see how spatial locality can affect performance.

Here's a simple test program that accesses a 1-dimensional tensor at different "strides". The "stride" of a sequence of accesses is just the distance between consecutive accesses. So, "stride 1" means accessing each element, and "stride 2" means accessing every other element. The outer loop ensures ensures that the total number of accesses the loop perform remains the same, regardless of stride size).

Examine the code below, and then answer these question:

In [ ]:

```
t = fiddle("stride.cpp", function="stride", name="spatial", opt="-O1",
code=r"""
#include"pin_tags.h"
#include"CNN/tensor_t.hpp"
#include"function_map.hpp"
#include<cstdint>

extern "C"
uint64_t* stride(uint64_t * data, uint64_t size, uint64_t arg1) {
    tensor_t<uint32_t> t(size,1,1,1, (uint32_t *)data);
    TAG_START("init", t.data, &t.as_vector(t.element_count()), true);

    for(uint i = 0; i < arg1; i++) {
        for(uint x = 0; x < size; x+=arg1) {
            t.get(x,0,0,0) = x;
        }
    }

    TAG_STOP("init");
    return data;
}

FUNCTION(one_array_1arg, stride);
""")
compare([t.source, t.cfg])
```

Question 13 (Completeness)

How many misses-per-instruction would you expect for `stride == 1` ? How about `stride == 8` ? Assume `size` is very large and that cache lines are 64 bytes.

`stride == 1` MPI:

`stride == 8` MPI:

`stride == 32` MPI:

Show Solution

Let's see if those predictions match reality. Run the cell below. It'll run the code above.

In [ ]:
```
t = fiddle("stride.cpp", function="stride", name="spatial", run=["perf_cou
           cmdline=f"--size {1024*1024*128} --arg1 1 2 4 8 16 32 64 --itei
           perf_cmdline="--stat-set L1.cfg --MHz 3500")
compare([t.source, t.cfg])
#show_trace(t.mtrace, show_tag="init")

display(render_csv("spatial.csv", columns=["function","size", "arg1", "L1_
plotPE("spatial.csv", lines=True, what=[("arg1", 'L1_MPI')])
```

In [ ]:
```
!cse142 job run --lab test './fiddle.exe --lib ./build/stride.so --functi
```

Question 14 (Completeness)

Do the measurements match our predictions? Does anything seem surprising about the results?

Of course, it's not really the misses we are worried about -- it's the impact on performance.

Recall what happens on a cache miss: Instead of having accessing the data in the cache, the processor must go down the memory hierarchy. In the worst case, this means going to main memory, which can easily take 100s of cycles. Let's imagine it's 200 cycles. What will the impact be on CPI?

Question 15 (Correctness - 2pts)

If a cache miss increases memory instruction latency by 200 cycles, how much should CPI increase between a stride of 1 and a stride of 64 (based on the data above)?

Here's how it actually played out:

In [ ]:
```
display(render_csv("spatial.csv", columns=["function","size", "arg1", "IC
plotPE("spatial.csv", lines=True, what=[("arg1", 'L1_MPI'),("arg1", 'CPI'
```

There's no particular reason to expect these number to match your calculation in the question above. We don't know exactly how long an L1 cache miss takes. 200 cycles is a guess, but in both cases the large impact of cache performance on execution time should be clear.

11.2 Temporal Locality

We'll discuss temporal locality in the next lab.

12 Programming Assignment

Your programming assignment for this lab is to implement a memory allocator. You've used memory allocators a lot: in C++ you access them via `new` and `delete`. In C, it's `malloc()` and `free()` (which `new` and `delete` call internally). The default implementation of `malloc()` and `free()` are general purpose: they are reasonably fast and can allocate data of any size. However, many applications need to allocate many, many objects very quickly, so a general-purpose allocator is too slow. In these cases, it makes sense to implement a specialized memory allocator because they can be much faster.

In this lab, you'll be implementing a specialized memory allocator that has these basic characteristics:

1. It can only allocate objects of a single size.
2. It will ensure those objects are aligned to configurable size.
3. The allocator will be an object that can allocate and deallocate objects using a simple interface (see below).
4. When the allocator is deleted, it will automatically deallocated all objects that are still allocated.

Here's the reference implementation:

In [18]:
```
render_code("ReferenceAllocator.hpp")
```

```
// ReferenceAllocator.hpp:1-49 (49 lines)
#include <stdlib.h>


template<
        class T,          // This is the type we are allocating.  You can
assume this is less than or equal to 4kB
        size_t ALIGNMENT  // The alignment at which we much allocate the
 objects.  You can assume this is less than or equal to 4kB
        >
class ReferenceAllocator {
        std::set<T*> chunks; // We store everything we allocated so we ca
n clean up in the destructor.
public:
        typedef T ItemType; // This will make T available as ReferenceAll
ocator::ItemType
        static const size_t Alignment = ALIGNMENT;  // Likewise, we can a
ccess the alignment as ReferenceAllocator::Alignment

        ReferenceAllocator() {}

        T * alloc() {
                void* p  = NULL;
                // this system call can allocate aribitrary-sized and ali
gned
                // objects.  Since it can handle any size, it's more gene
ral.
                int r =  posix_memalign(&p, ALIGNMENT, sizeof(T));
                if (r == -1) {
                        std::cerr << "posix_memalign() failed.  Exiting:
 " << strerror(errno) << "\n";
                        exit(1);
                }
                uint8_t * t = reinterpret_cast<uint8_t*>(p); // alloc_chu
nk provides void*, but we can assign to void.  So cast...
                for(uint i= 0; i < sizeof(T); i++) {
                        t[i] = 0; // and set to zero.
                }
                T* c = reinterpret_cast<T*>(p); // cast to the type we al
locate.
                new (c) T; // This is the "in place" new operator.  It co
nstructs an object at a given location.
                chunks.insert(c); // record it so we can delete it later.
                return c;
        }
```

```
        void free(T * p) {
                std::free(reinterpret_cast<void*>(p)); // Return the memo
ry
                chunks.erase(p); // note that it's no longer allocated.
        }

        ~ReferenceAllocator() {
                for(auto & p: chunks) {
                        std::free(reinterpret_cast<void*>(p)); // Return
 everything that still allocated.
                }
        }
};

template<class T, size_t ALIGNMENT>
const size_t ReferenceAllocator<T, ALIGNMENT>::Alignment;
```

First, note that it's a template class that takes two parameters:

- `T` : the type it will allocate. You can assume the size of `T` can be up to 4096 bytes.
- `ALIGNMENT` : The alignment of the allocations it will make. Alignment size can be powers up 2 up to 4096 bytes (1, 2, 4, 8, 16, etc.).

The allocator has just four methods:

- `ReferenceAllocator()` : the constructor.
- `alloc()` allocates and returns an instance of `T` .
- `free()` deallocates `p` , a previously allocated instance of `T` .
- `~ReferenceAllocator()` : The destructor for the allocator. It needs to clean up all the memory the allocator manages.

It also defines `ReferenceAllocator::ItemType` , which let's us access the type the allocator allocates and `ReferenceAllocator::Alignment` which gives us access to the alignment size.

The implementation above just relies on the standard library's alignment-aware interface to the general-purpose allocator. This means it's not optimized to exploit the fact that we only need to allocate a single size of object, and this is where the big optimization opportunities lie.

Your allocator will not rely on any of the normal memory-allocating library calls. Instead will use a very simple allocator to allocate memory in bulk from the operating system. The interface is called `ChunkAlloc` :

In [19]:
```
render_code("ChunkAlloc.hpp")
render_code("ChunkAlloc.cpp")
```

```
// ChunkAlloc.hpp:1-7 (7 lines)
#define CHUNK_SIZE (128*1024)

extern void init_chunk(); // Set things up.
extern void * alloc_chunk(); // Allocate CHUNK_SIZE of memory
extern void free_chunk(void*p); // Free CHUNK_SIZE of memory
extern size_t get_allocated_chunks(); // Query how many chunks are curren
tly allocated.
```

```
// ChunkAlloc.cpp:1-43 (43 lines)
#include <stdlib.h>
#include<iostream>
#include <string.h>
#include "ChunkAlloc.hpp"
#include <sys/mman.h>
#include"pin_tags.h"

static size_t allocated_chunks = 0;

void init_chunk() {
        // This creates an tag that covers no memory, since the max is ve
ry small and min is very large.
        // We'll grow it below.
        TAG_START("chunks", reinterpret_cast<void*>(-1), reinterpret_cast
<void*>(0), true);
}

void * alloc_chunk() { // allocate CHUNK_SIZE bytes of memory by asking t
he operating system for it.

        // this is actually malloc gets it's memory from the kernel.
        // mmap() can do many things.  In this case, it just asks the ker
nel to
        // give us some pages of memory.  They are guaranteed to contain
 zeros.
        void * r = mmap(NULL, CHUNK_SIZE, PROT_READ|PROT_WRITE, MAP_SHARE
D|MAP_ANONYMOUS, 0, 0);
        if (r == MAP_FAILED) {
                std::cerr << "alloc_chunk() failed. This often means yo
u've allocated too many chunks. Exiting: " << strerror(errno) << "\n";
                exit(1);
        }
        TAG_GROW("chunks", r, reinterpret_cast<uint8_t*>(r) + CHUNK_SIZE
```

```
);
        allocated_chunks++; // This is just statistics tracking
        return r;
}

void free_chunk(void*p) { // Return the chunk to the OS.  After this, acc
esses to the addresses in the chunk will result in SEGFAULT
        int r = munmap(p, CHUNK_SIZE);
        if (r != 0) {
                std::cerr << "free_chunk() failed. exiting: " << strerror
(errno) << "\n";
                exit(1);
        }
        allocated_chunks--;
}

size_t get_allocated_chunks() {
        return allocated_chunks;
}
```

`ChunkAlloc` gets memory the same way `malloc()` does: By calling the `mmap()` system call which stands for "memory map". `mmap()` is a great tool and can do many things. You can read about it here (https://man7.org/linux/man-pages/man2/mmap.2.html), but it's not necessary for the lab. What is important for our purposes is that `alloc_chunk()` will return a 128kB region of memory that is 4kB-aligned.

12.1 Detailed Requirements

You're going to build your own version of `ReferenceAllocator` called `SolutionAllocator`. You'll find a copy of `ReferenceAllocator.hpp` in `SolutionAllocator.hpp`. Do your work there.

Now that you understand the basics of how `ReferenceAllocator` works, here's the detailed list of requirements for `SolutionAllocator`:

1. All addresses that `SolutionAllocator::alloc()` returns must be aligned to `ALIGNMENT` so `addr % ALIGNMENT == 0`.
2. All addresses that `SolutionAllocator::alloc()` returns must point to at least `sizeof(T)` bytes of memory.
3. `SolutionAllocator::alloc()` needs to set all bytes of memory that the instance of `T` will occupy to zero before constructing `T`.
4. `SolutionAllocator::alloc()` needs to construct an instance of `T` in the memory using the in-place `new` operator (see below).
5. `SolutionAllocator::alloc()` cannot return the same pointer twice unless the pointer has been deallocated with `SolutionAllocator::free()` first.
6. After the destructor completes, `SolutionAllocator` must have called `free_chunk()` for every chunk it allocated with `alloc_chunk()`. I.e., `get_allocated_chunks()` must return 0.

7. You are free to use the STL data structures for the internals of `SolutionAllocator`, but `alloc()` may not return any memory that is a part of an STL container.
8. The only mechanism you can use to allocate memory is `alloc_chunk()/free_chunk()`. No calls to `malloc()` (or other functions from the standard library that allocate raw memory) or `new` (other than the "in place" version.) to allocate the space `SolutionAllocator` will return.
9. Your allocator must recycle: If memory is returned to it via `free()`, your allocator should reallocate that memory before requesting new memory via `alloc_chunk()`. This prevents your allocator from continually allocating new memory, which is very fast, but not a realistic solution.

`ReferenceAllocator` already satisfies all of these except the last two: `ReferenceAllocator` uses `posix_memalign()` which is forbidden in your solution. With respect to recycling, I'm not sure how `posix_memalign()` works internally, so I'm not sure how it recycles, but it does something reasonably efficient. You'll find that removing `posix_memalign()` and meeting the above criteria will require you to rewrite most of the code in `SolutionAllocator.hpp`.

12.2 Evaluation

Your implementation will be evaluated based on correctness and performance. Your implementation of `SolutionAllocator` must pass the tests in `run_test.cpp` which cover the requirements listed above.

The performance portion is based on the average score over several benchmarks in `Allocator.cpp`. Details of the grade calculation are given under "Final Measurement" below.

The code for the benchmarks is in `Allocator.cpp`. The code is below. The key functions are `bench()`, `microbench()`, `exercise()`, and `miss_machine()`. Here's what they do:

- `microbench()` just calls `alloc()` many times and records the execution time. Then it does the same for `free()`.
- `exercise()` allocates a bunch of objects, deletes some at random, allocates some more, deletes some at random, etc. This puts your allocator into a "warmed up" or "well-used" state to approximate how it would behave in a long-running program.
- `bench()` measure the execution time of `exercise()`.
- `miss_machine()` measures the speed of a specialized version of the miss machine that tests how well your allocator manages spatial locality (more below)

The `run_*` functions near the bottom are wrappers to record information about the parameters used to run each benchmark. The functions marked `extern "C"` at the bottom of `Allocator.cpp` allow us to invoke the benchmarks from the command line and run the tests with a variety of parameters.

In [20]:
```
render_code("Allocator.cpp")
```

```
// Allocator.cpp:1-251 (251 lines)
#include <stdlib.h>
#include<iostream>
#include<archlab.hpp>
#include"function_map.hpp"
#include<map>
#include"fast_URBG.hpp"
#include"ReferenceAllocator.hpp"
#include"SolutionAllocator.hpp"
#include"BonusAllocator.hpp"
#include"pin_tags.h"

template<class Allocator>
void exercise(Allocator * allocator, size_t count, int iterations, uint64
_t seed, bool cleanup = false) {
        // Interesting allocator behaviors an bugs emerge when the alloca
tor
        // has to allocate and free objects in complex patterns.
        //
        // To simulate that, we allocate count items and then, on each
        // iteration, free about 1/4 of them and replace them with new it
ems.

        std::vector<typename Allocator::ItemType *> items(count);
        TAG_START_ALL("exercise", true);

        for(unsigned int i = 0; i < count; i++)
                items[i] = NULL;

        for(int i = 0; i < iterations; i++) {
                for(unsigned int j = 0; j < count; j++) {
                        if (items[j] == NULL) {
                                items[j] = allocator->alloc();
                        }
                }
                for(unsigned int j = 0; j < count; j++) {
                        fast_rand(&seed);
                        if (seed & 0x3) {
                                allocator->free(items[j]);
                                items[j] = NULL;
                        }
                }
        }
        if (cleanup) {
                for(unsigned int j = 0; j < count; j++) {
```

```
                        if (items[j]) {
                                allocator->free(items[j]);
                                items[j] = NULL;
                        }
                }
        }

        TAG_STOP("exercise");
}


template<class Allocator>
void bench(uint64_t count, uint64_t seed, bool do_exercise) {
        {
                auto alloc = new Allocator;
                if (do_exercise){ // warm it up.
                        exercise<Allocator>(alloc, 4000, 20, seed);
                }
                ArchLabTimer timer;
                timer.attr("count", count);
                timer.attr("seed", seed);
                timer.attr("test", "exercise");
                timer.attr("exercise", do_exercise);
                timer.go();
                exercise<Allocator>(alloc, count/16, 16, seed);
                delete alloc;
        }
}

template<class Allocator>
void microbench(uint64_t count, uint64_t seed, bool do_exercise) {

        auto alloc = new Allocator;

        if (do_exercise) { // get the allocator warmed up.
                exercise<Allocator>(alloc, 4000, 20, seed);
        }
        std::vector<typename Allocator::ItemType*> items(count);
        {
                ArchLabTimer timer;
                timer.attr("count", count);  // These show up in the csv
 file.
                timer.attr("seed", seed);
                timer.attr("test", "alloc");
                timer.attr("exercise", do_exercise);
                timer.go(); // start timing here
                for(uint64_t i = 0; i < count; i++) {
                        items[i] = alloc->alloc();
                }
```

```
        }

        if (do_exercise) {
                exercise<Allocator>(alloc, 4000, 20, seed);
        }
        {
                ArchLabTimer timer;
                timer.attr("count", count);
                timer.attr("seed", seed);
                timer.attr("test", "free");
                timer.attr("exercise", do_exercise);
                timer.go();
                for(uint64_t i = 0; i < count; i++) {
                        alloc->free(items[i]);
                }
        }
        delete alloc;

}

//BEGIN
struct MissingLink {
        struct MissingLink * next;
};

extern "C"
struct MissingLink*  __attribute__((noinline)) do_misses(struct MissingLi
nk * l) {
        for(uint i = 0; i < 10000000; i++) {
                l = l->next;
        }
        return l;
}

template<class Allocator>
uint64_t  miss_machine(uint64_t count, uint64_t seed) {
        auto alloc = new Allocator; // create the allocator.

        exercise<Allocator>(alloc, 10000, 20, seed); // warm it up.

        TAG_START_ALL("build_miss_machine", true); // For Moneta tracing

        std::vector<struct MissingLink *> links(count);  // Storage for t
he links
        for(auto &i : links) {                           // allocate the
m.
                i = alloc->alloc();
                i->next = NULL;
        }
```

```
        std::shuffle(links.begin(), links.end(), fast_URBG(seed));  // ra
ndomize the order of the links
        for(uint i = 0; i < links.size() -1; i++) {
                links[i]->next = links[i+1];            // Make the next p
ointers reflect the ordering.
        }
        links.back()->next = links.front(); // complete the circle

        struct MissingLink * l = links[0];
        {
                ArchLabTimer timer;
                timer.attr("count", count); // record data about experime
nt
                timer.attr("seed", seed);
                timer.attr("test", "miss_machine");
                timer.go();

                TAG_STOP("build_miss_machine");// For Moneta tracing

                TAG_START_ALL("miss_machine", true); // for moneta tracin
g
                l = do_misses(l);        // Do the misses.  Put it in a f
unction so we can easily look at the CFG.
                TAG_STOP("miss_machine"); // for moneta tracing
        }
        return reinterpret_cast<uintptr_t>(l); // Return something  depen
ding on do_misses() to prevent the compiler from optimizing it away.
}
//END


template<class Allocator>
void run_bench(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("bytes", sizeof(typename Allocator
::ItemType));
        theDataCollector->register_tag("alignment", Allocator::Alignment
);
        bench<Allocator>(count, seed, true);
}

template<class Allocator>
void run_miss_machine(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("test", "run_miss_machine");
        theDataCollector->register_tag("bytes", sizeof(typename Allocator
::ItemType));
        theDataCollector->register_tag("alignment", Allocator::Alignment
);
        miss_machine<Allocator>(count, seed);
```

```
}

template<class Allocator>
void run_microbench(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("bytes", sizeof(typename Allocator
::ItemType));
        theDataCollector->register_tag("alignment", Allocator::Alignment
);
        microbench<Allocator>(count, seed, true);
}



// Starter functions

extern "C"
uint64_t * allocator_bench_starter(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("impl", "ReferenceAllocator");
        run_bench<ReferenceAllocator<uint8_t[3], 16>>(count, seed);
        run_bench<ReferenceAllocator<uint8_t[125], 32>>(count, seed);
        run_bench<ReferenceAllocator<uint8_t[4096], 4096>>(count, seed);
        return NULL;
}
FUNCTION(alloc_test, allocator_bench_starter); // this let's you pass thi
s function as an argument to `--function` on the command line.


extern "C"
uint64_t * allocator_microbench_starter(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("impl", "ReferenceAllocator");
        run_microbench<ReferenceAllocator<uint[4], 8>>(count, seed);
        run_microbench<ReferenceAllocator<uint[1024], 4096>>(count, seed
);
        return NULL;
}
FUNCTION(alloc_test, allocator_microbench_starter);


extern "C"
uint64_t * miss_machine_starter(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("impl", "ReferenceAllocator");
        run_miss_machine<ReferenceAllocator<struct MissingLink, sizeof(st
ruct MissingLink)>>(count, seed);
        return NULL;
}
FUNCTION(alloc_test, miss_machine_starter);


// Solution functions
```

```
extern "C"
uint64_t * allocator_bench_solution(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("impl", "SolutionAllocator");

        run_bench<SolutionAllocator<uint8_t[3], 16>>(count, seed);
        run_bench<SolutionAllocator<uint8_t[125], 32>>(count, seed);
        run_bench<SolutionAllocator<uint8_t[4096], 4096>>(count, seed);

        return NULL;
}
FUNCTION(alloc_test, allocator_bench_solution);

extern "C"
uint64_t * allocator_microbench_solution(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("impl", "SolutionAllocator");
        run_microbench<SolutionAllocator<uint[4], 8>>(count, seed);
        run_microbench<SolutionAllocator<uint[1024], 4096>>(count, seed);
        return NULL;
}
FUNCTION(alloc_test, allocator_microbench_solution);

extern "C"
uint64_t * miss_machine_solution(uint64_t count, uint64_t seed) {
        theDataCollector->register_tag("impl", "SolutionAllocator");
        run_miss_machine<SolutionAllocator<struct MissingLink, sizeof(str
uct MissingLink)>>(count, seed);
        return NULL;
}
FUNCTION(alloc_test, miss_machine_solution);
```

12.3 `miss_machine()`

`miss_machine()` is the most interesting of the three benchmark functions because it addresses one of the subtle problems that can arise with a memory allocator.

Because the memory allocator controls the layout of a program's memory, it can have a strong impact on how a program accesses memory. For instance:

1. If the allocator is poorly designed (or unlucky) it might arrange memory so that there are many conflict misses.
2. The depending on how effectively the allocator reclaims space that is freed, it may waste space.
3. How the allocator places objects can affect which objects are close enough to benefit from spatial locality.

The `miss_machine()` function in `Allocator.cpp` explores the third issue. Here's the code again:

In [21]:
```
render_code("Allocator.cpp", show=("//BEGIN", "//END"))
```

```
// Allocator.cpp:110-159 (50 lines)
//BEGIN
struct MissingLink {
        struct MissingLink * next;
};

extern "C"
struct MissingLink*  __attribute__((noinline)) do_misses(struct MissingLi
nk * l) {
        for(uint i = 0; i < 10000000; i++) {
                l = l->next;
        }
        return l;
}

template<class Allocator>
uint64_t  miss_machine(uint64_t count, uint64_t seed) {
        auto alloc = new Allocator; // create the allocator.

        exercise<Allocator>(alloc, 10000, 20, seed); // warm it up.

        TAG_START_ALL("build_miss_machine", true); // For Moneta tracing

        std::vector<struct MissingLink *> links(count);  // Storage for t
he links
        for(auto &i : links) {                           // allocate the
m.
                i = alloc->alloc();
                i->next = NULL;
        }

        std::shuffle(links.begin(), links.end(), fast_URBG(seed));  // ra
ndomize the order of the links
        for(uint i = 0; i < links.size() -1; i++) {
                links[i]->next = links[i+1];            // Make the next p
ointers reflect the ordering.
        }
        links.back()->next = links.front(); // complete the circle

        struct MissingLink * l = links[0];
        {
                ArchLabTimer timer;
                timer.attr("count", count); // record data about experime
nt
                timer.attr("seed", seed);
```

```
                timer.attr("test", "miss_machine");
                timer.go();

                TAG_STOP("build_miss_machine");// For Moneta tracing

                TAG_START_ALL("miss_machine", true); // for moneta tracin
g
                l = do_misses(l);        // Do the misses.  Put it in a f
unction so we can easily look at the CFG.
                TAG_STOP("miss_machine"); // for moneta tracing
        }
        return reinterpret_cast<uintptr_t>(l); // Return something  depen
ding on do_misses() to prevent the compiler from optimizing it away.
}
//END
```

Here's what the benchmark does.

1. It create an allocator and warms it up with `exercise()`.
2. It builds a miss machine out of links allocated one-at-a-time from your allocator. This is different than description of the miss machine given earlier in the lab: In that case we allocated the links in an array, guaranteeing good spatial locality.
3. Then it measures the execution time of a call to `do_misses()`, which traverses the miss machine.

Read through the code and comments carefully to understand it.

We are going to run `miss_machine` with `count = 4096`. Since each `MissingLink` is 8 bytes, 4096 links *should* fit in the L1 cache, since `4096*8 = 32*1024`. Let's see how the allocators do:

In [ ]:
```
:e alloc_main.exe
:142 job run --lab caches --force "./alloc_main.exe --size 4096 --stats mis
```

In [ ]:
```
render_csv("miss_machine.csv", columns=["function", "count", "ET", "L1_MP
```

I get an `L1_MPI` of 0.143. Let's take a look at the CFG for `do_misses()`:

In [ ]:
```
do_cfg("alloc_main.exe", symbol="do_misses")
```

As you can see, there are only 5 instructions and 3 of them are accesses. However, two of them are copying the same value to and from the stack over and over (how inefficient! you should fix that!), so they will almost always hit. That leave 1 load which accounts for 0.2 of the instructions in the

loop body. Since MPI of 0.14, means that about 0.14/0.2 = 0.7 (or 70%) of the loads are missing.

Your task is to speed up `do_misses()` and the best way to do that is to reduce MPI by allocating `MissingLinks` in such a way that you maximize spatial locality across the miss machine.

A couple of things to keep in mind:

1. This benchmark is only 12.5% of your grade, and your work on the other benchmarks will get you most of those points. It's also the hardest part. Work on it last.
2. A locality-aware allocator is probably more complex than a non-locality-aware allocator. If you add too much complexity, it will reduce performance on the other benchmarks.
3. Remember that the key to spatial locality is to fit your data (in this case the links in the miss machine) into the smallest possible number of cache lines possible.

12.4 Useful C++

C++ likes to keep you safe by not letting you convert pointers to integers or letting you convert pointers from one type to another. However, memory allocators need to break the rules occasionally to transform untyped bytes into objects. The main tool for this is `reinterpret_cast<>()` and the "in place" `new` operator. `ReferenceAllocator` provides an example of how to use both mechanisms.

12.4.1 `reinterpret_cast<>()`

`reinterpret_cast<>()` let's you change a values from one type to another as long as they are the same size. So you can do this:

```
int *x;
char *y = reinterpret_cast<int *>(x);

// or

void * x = alloc_chunk();
T * t = reinterpret_cast<T*>(x);
```

If you are familiar with C's `(T*)(x)` casting syntax, `reinterpret_cast` is equivalent, but preferred because it's easier to spot in code.

A related tool is `uintptr_t` , which is an unsigned integer that is the same size as a pointer. So you can increment a pointer by one byte doing this:

```
int * x = new int;
uintptr_t n = reinterpret_cast<uintptr_t>(x);
n += 1;
x = reinterpret_cast<int *>(n);
```

Note that this is different than;

```
int *x = new int;
x++;
```

Since, under C++'s pointer arithmetic rules, if you increment a pointer of type `T*`, it actually increases the address by `sizeof(T)` (i.e., 4 bytes for an `int`).

12.4.2 In-Place `new`

If you call `new T`, C++ will allocate some memory to hold a new instance of `T` and then run `T`'s constructor on it. But what if *you* want to decide where to construct the new instance?

In that case you can say something like this:

```
void * p = alloc_chunk();
new (p) T;
```

To construct a new instance of `T` "in place" in the memory pointed to by `p`. Or, if you wanted to initialize an instance of `T` starting at the 11th byte after `p`, you could do this:

```
uintptr_t n = reinterpret_cast<uintptr_t>(p);
n += 11;
void *q = reinterpret_cast<void*>(n);
new (q) T;
```

Why would anyone ever want to do that?

12.5 How To Do This Lab

Here's some tips about how to approach this lab.

12.5.1 The Lifecycle of a Memory Allocator

Here are the basic steps that your memory allocator must accomplish over it's lifetime.

1. Initialize itself and its internal data structures.
2. Respond to calls to `alloc()`
   A. If the allocator has recycled objects, initialize one and return it.
   B. Otherwise, if the allocator has new (not recycled) memory on hand, initialize one object worth, and return it.
   C. Otheriwe, if the allocator has no memory on hand, call `alloc_chunk()`, and goto 2.2.
3. Respond to calls to `free()`
   A. Store the object somewhere so it can be recycled.
4. On destruction, called `free_chunk()` to deallocate all the memory you allocated with `alloc_chunk()`.

12.5.2 Thing to Try

Here are some ideas about how to get started:

1. The main difference between what you'll implement in `SolutionAllocator` and `posix_memalign()` is that `SolutionAllocator` only needs to allocate objects of a single size. You can exploit this fact to improve performance.

2. `alloc_chunk()` let's you allocate enough space to store many objects. Since the objects are all the same size and alignment, you can calculate where each of instance of `T` will reside with the chunk.
3. You need to recycle. So, think about how you can efficiently store `free()` ed memory while it's waiting to be re-`alloc()` ed.
4. You will need to choose the right data structures and algorithms to achieve good performance. Think about what each data structure *needs* to do and what operations are most important to performance. Use the STL! (In Lab 2, I noticed many students implementing things that the STL already provides. Don't re-invent -- or debug -- the wheel!)

Your overall score is based on your allocator's performance across eight benchmarks. This might seem daunting, but the performance of the benchmarks is very correlated: If you speed up your allocator for one of them, it will get faster for many of the others.

With that in mind, start with the simplest ones and go from there. I'd proceed in this order:

1. The `microbench` function.
2. The `bench` function.
3. The `miss_machine` function.

12.6 Do Your Work Here [...]

Below are the key commands you'll need to make progress on the lab. Your solution should go in `SolutionAllocator.hpp`:

12.7 Tools [...]

These are some tools you might find useful as you optimize your implementation. I encourage you to give some of them a try.

12.8 Final Measurement

When you are done, make sure your best allocator is called `SolutionAllocator` in `SolutionAllocator.hpp`. Then you can submit your code to the Gradescope autograder. It will run the commands given above and compute your grade.

Your grade is based on your speed up relative `ReferenceAllocator` on the eight benchmarks.

For each of them, there's a target speedup given the table below. You get a score for each benchmark between 0 and 12.5, and the overall score is the sum of these scores. For each function, the score is compute as `your_speedup/target_speedup * 12.5`.

For this lab, you don't get extra credit for beating the targets. This will help ensure that your design in balanced: You must do well at all 8 benchmarks to do well on the lab.

To get points, your code must also be correct. The autograder will run the regressions in `run_tests.cpp` to check it's correctness.

You can mimic exactly what the autograder will do with the command below, and then run the next cell below to list them and the target speedups.

After you run it, the results will be in `autograde/bench.csv`, `autograde/microbench.csv`, and `autograde/miss_machine.csv` rather than `./bench.csv`, `./microbench.csv`, and `miss_machine.csv`. This command builds and runs your code in a more controlled way by doing the following:

1. Ignores all the files in your repo except `SolutionAllocator.cpp` and `config.make`.
2. Copies those files into a clean clone of the starter repo.
3. Builds `alloc_main.exe` from scratch.
4. And then runs the commands for the benchmarks.
5. It then runs `autograde.py` to compute your grade.

Running the cell below will do the same thing as the Geradescope autograder. And the cell below shows the name and target speedups for each benchmark. This takes 1-2 minutes to run.

In [ ]:
```
!make alloc_main.exe
!cse142 job run --take SolutionAllocator.hpp --take config.make --lab cac
```

You can check the performance results like this:

In [25]:
```
!./autograde.py --submission autograde --results autograde.json
from autograde import compute_all_scores
df = compute_all_scores(dir="autograde")[0]
display(df)
print(f"total points: {round(sum(df['capped_score']), 2)}/100")
```

	label	test	bytes	alignment	target_speedup	baseline_ET	ET	spe
0	allocator_microbench_solution alloc 16 8	alloc	16	8	52.00	0.2550	0.255951	0.99
1	allocator_microbench_solution free 16 8	free	16	8	60.60	0.2120	0.213061	0.99
2	allocator_microbench_solution alloc 4096 4096	alloc	4096	4096	7.32	3.6780	3.669836	1.00
3	allocator_microbench_solution free 4096 4096	free	4096	4096	73.22	0.2770	0.277798	0.99
0	allocator_bench_solution 3 16	exercise	3	16	15.24	0.1030	0.102434	1.00
1	allocator_bench_solution 125 32	exercise	125	32	19.40	0.1420	0.142694	0.99
2	allocator_bench_solution 4096 4096	exercise	4096	4096	27.27	1.0690	1.057674	1.01
0	miss_machine	miss_machine	8	8	4.96	0.0897	0.097012	0.92

```
total points: 6.58/100
```

The "capped_score" column contains the number of points you'll receive.

You can also inspect the autograder's output.

In [ ]:
```
render_code("autograde.json")
```

Most of it is internal stuff that gradscope needs, but the key parts are the `score`, `max_score`, and `output` fields.

All that's left is commit your code:

In [ ]:
```
!git commit -am "Solution to the lab."
!git push
```

If `git commit` tell you something like:

```
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'prcheng@dsmlp-jupy
ter-prcheng.(none)')
Warning: Permanently added the RSA host key for IP address '140.82.
112.3' to the list of known hosts.
Everything up-to-date
```

Then you can do (but fill in your @ucsd.edu email and your name):

In [ ]:
```
!git config --global user.email "you@example.com"
!git config --global user.name "Your Name"
```

13 Recap

In this lab, you have seen how important memory alignment is and how the compiler lays out structs to enforce it. You have also started to learn how to write cache-aware code and to think in terms of cache lines and cache misses rather than bytes and memory accesses as you reason about how your programs access memory. You have measured spatial locality and seen how its presence or absence can affect performance. You have also implemented a memory allocator and seen how a memory allocator can affect the spatial locality of the data structures it allocates.

14 Turning In the Lab [...]