So, how can the memory system arrange for the right data to be in the right place at the right time?

Our goal is to have the frequently-used data in some fast SRAM.

That means the memory system will have to be able to predict which memory locations will be accessed.

And to keep the overhead of moving data into and out of SRAM manageable, we'd like to amortize the cost of the move over many accesses.

In other words we want any block of data we move into SRAM to be accessed many times.

When not in SRAM, data would live in the larger, slower DRAM that serves as main memory.

If the system is working as planned, DRAM accesses would happen infrequently, e.g., only when it's time to bring another block of data into SRAM.

If we look at how programs access memory, it turns out we *can* make accurate predictions about which memory locations will be accessed.

The guiding principle is "locality of reference" which tells us that if there's an access to address X at time t, it's very probable that the program will access a nearby location in the near future.

To understand why programs exhibit locality of reference, let's look at how a running program accesses memory.

Instruction fetches are quite predictable.

Execution usually proceeds sequentially since most of the time the next instruction is fetched from the location after that of the current instruction.

Code that loops will repeatedly fetch the same sequence of instructions, as shown here on the left of the time line.

There will of course be branches and subroutine calls that interrupt sequential execution, but then we're back to fetching instructions from consecutive locations.

Some programming constructs, e.g., method dispatch in object-oriented languages, can produce scattered references to very short code sequences (as shown on the right of the time line) but order is quickly restored.

This agrees with our intuition about program execution.

For example, once we execute the first instruction of a procedure, we'll almost certainly execute the remaining instructions in the procedure.

So if we arranged for all the code of a procedure to moved to SRAM when the procedure's first instruction was fetched, we'd expect that many subsequent instruction fetches could be satisfied by the SRAM.

And although fetching the first word of a block from DRAM has relatively long latency, the DRAM's fast column accesses will quickly stream the remaining words from sequential addresses.

This will amortize the cost of the initial access over the whole sequence of transfers.

The story is similar for accesses by a procedure to its arguments and local variables in the current stack frame.

Again there will be many accesses to a small region of memory during the span of time we're executing the procedure's code.

Data accesses generated by LD and ST instructions also exhibit locality.

The program may be accessing the components of an object or struct.

Or it may be stepping through the elements of an array.

Sometimes information is moved from one array or data object to another, as shown by the data accesses on the right of the timeline.

Using simulations we can estimate the number of different locations that will be accessed over a particular span of time.

What we discover when we do this is the notion of a "working set" of locations that are accessed repeatedly.

If we plot the size of the working set as a function of the size of the time interval, we see that the size of the working set levels off.

In other words once the time interval reaches a certain size the number of locations accessed is approximately the same independent of when in time the interval occurs.

As we see in our plot to the left, the actual addresses accessed will change, but the number of *different* addresses during the time interval will, on the average, remain relatively constant and, surprisingly, not all that large!

This means that if we can arrange for our SRAM to be large enough to hold the working set of the program, most accesses will be able to be satisfied by the SRAM.

We'll occasionally have to move new data into the SRAM and old data back to DRAM, but the DRAM access will occur less frequently than SRAM accesses.

We'll work out the mathematics in a slide or two, but you can see that thanks to locality of reference we're on track to build a memory out of a combination of SRAM and DRAM that performs like an SRAM but has the capacity of the DRAM.

The SRAM component of our hierarchical memory system is called a "cache".

It provides low-latency access to recently-accessed blocks of data.

If the requested data is in the cache, we have a "cache hit" and the data is supplied by the SRAM.

If the requested data is not in the cache, we have a "cache miss" and a block of data containing the requested location will have to be moved from DRAM into the cache.

The locality principle tells us that we should expect cache hits to occur much more frequently than cache misses.

Modern computer systems often use multiple levels of SRAM caches.

The levels closest to the CPU are smaller but very fast, while the levels further away from the CPU are larger and hence slower.

A miss at one level of the cache generates an access to the next level, and so on until a DRAM access is needed to satisfy the initial request.

Caching is used in many applications to speed up accesses to frequently-accessed data.

For example, your browser maintains a cache of frequently-accessed web pages and uses its local copy of the web page if it determines the data is still valid, avoiding the delay of transferring the data over the Internet.

Here's an example memory hierarchy that might be found on a modern computer.

There are three levels on-chip SRAM caches, followed by DRAM main memory and a flash-memory cache for the hard disk drive.

The compiler is responsible for deciding which data values are kept in the CPU registers and which values require the use of LDs and STs.

The 3-level cache and accesses to DRAM are managed by circuity in the memory system.

After that the access times are long enough (many hundreds of instruction times) that the job of managing the movement of data between the lower levels of the hierarchy is turned over to software.

Today we're discussing how the on-chip caches work.

In Part 3 of the course, we'll discuss how the software manages main memory and non-volatile storage devices.

Whether managed by hardware or software, each layer of the memory system is designed to provide lower-latency access to frequently-accessed locations in the next, slower layer.

But, as we'll see, the implementation strategies will be quite different in the slower layers of the hierarchy.