Memory and Locality

So far we've been talking about performance as though all steps in the evaluation model are equally expensive. This turns out not to be even approximately true; they can differ by two orders of magnitude or more. Asymptotically, it doesn't matter, because there is an upper bound on how long each step takes. But in practice, a 100-fold slowdown can matter if it's affecting the performance-critical part of the program. To write fast programs, we sometimes need to understand how the performance of the underlying computer hardware affects the performance of the programming language, and to design our programs accordingly.

The memory abstraction

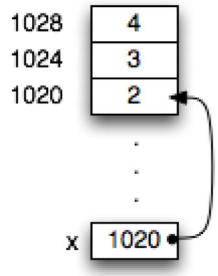
Computers have a memory, which can be abstractly thought of as a big array of ints, or more precisely, of 32-bit **words**. The indices into this array are called addresses. Depending on the processor, addresses may be either 32 bits long (allowing up to 4 gigabytes to be addressed) or 64 bits, allowing a memory four billion times larger. The operations supported by the memory are reading and writing, analogous to `Array.sub` and `Array.update`:

```
module type MEMORY = sig
  type address
  (* Returns: read(addr) is the current contents of memory at addr. *)
  val read: address -> int
  (* Effects: write(addr, x) changes memory at addr to contain x. *)
  val write: address * int -> unit
end
```

All of the rich features and types of a higher level language are implemented on top of this simple abstraction. For example, every variable in the program has, when it is in scope, a location in memory assigned to it. (Actually the compiler tries to put as many variables as possible into `registers`, which are much faster than memory, but we can think of these as just other memory locations for the purpose of this discussion.) For example, if we declare `let x: int = 2`, then at run time there is a memory location for x, which we can represent as a box containing the number 2:

Boxes and pointers

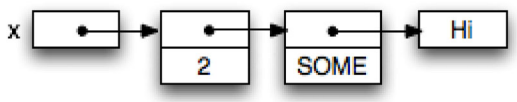
Since a variable box can only hold 32 bits, many values cannot fit in the location of their variable. Therefore the language implementation puts such values elsewhere in memory. For example, a tuple `let x = (2,3,4)` would be represented as three consecutive memory locations, which we can draw as boxes. The memory location (box) for x would contain the address of the first of these memory locations. We can think of this address as an arrow pointing from the box for x to the first box of the tuple. Therefore we refer to this address as a **pointer**. Since the actual memory addresses don't usually matter, we can draw diagrams of how memory is being used that omit the addresses and just show the pointers.



Meta Platforms, Inc.
Exhibit 1059

A tuple type is an example of a **boxed type**: its values are too large to fit inside the memory location of a variable, and so need their own memory locations, or boxes. Unboxed types in OCaml include the basic types such as int, char, bool, real, nil, and NONE. Boxed types include tuples, records, datatypes, list nodes, arrays, refs, strings, and functions.

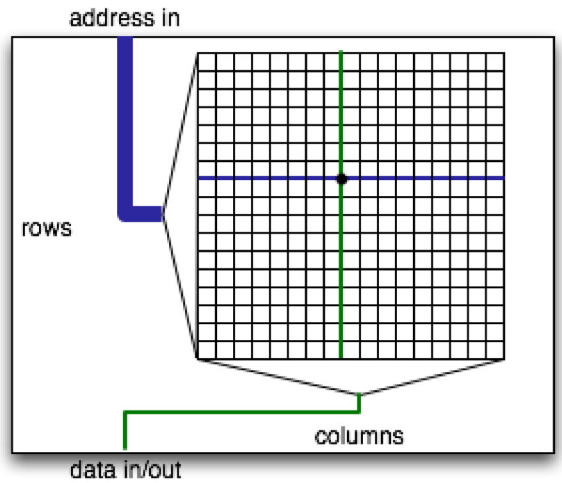
Every time we have a variable name with a boxed type, it means that there to access that value, the processor must use a pointer address to get to the data. Following pointers is called an **indirection**. For example, if we declare a variable `let x = (2, Some "Hi")`, three indirections are needed to get to the characters of the string "Hi":



The reason we care about whether there are pointers or not is that indirections can be very expensive. To understand why, we need to look at how computer memory works more carefully.

The memory implementation

The memory of a computer is made of a bunch of big arrays etched into various chips. Each array has a large number of row and column lines that the memory chip can independently activate. At the intersection of each row and column line, there is a a bit of storage implemented using one or more transistors.



To read bits out of a row of memory, each transistor on the row has to pump enough electrons onto or off of its column line to allow sensing logic on the column to detect whether that bit was 0 or 1. All else equal, bigger memories are slower because the transistors are smaller relative to the column line they have to activate. As a result, computer memory has gotten slower and slower relative to the speed of processing (e.g., adding two numbers). Adding two numbers takes one cycle (or even less on multiple-issue machines such as the x86 family). A typical computer memory these days takes about 30ns to deliver requested data. This sounds pretty fast, but you have to keep in mind that a 4GHz processor is doing a new instruction every 0.25ns. Thus, the processor will have to wait more than 100 cycles every time the memory is needed to deliver data. Quite a bit of computation can be done in the time it takes to fetch one number from memory.

Caches

To deal with the relative slowdown of computer memory, computer architects have introduced **caches**, which are smaller, faster memories that sit between the CPU and the main memory. A cache keeps track of the contents of memory locations recently requested by the processor. If the processor asks for one of these locations, the cache

Meta Platforms, Inc.
Exhibit 1059 2/5
Page 002

gives the answer instead. Because the cache is much smaller than main memory (hundreds of kilobytes instead of tens or hundreds of megabytes), it can be made to deliver requests much faster than main memory: in tens of cycles rather than hundreds. In fact, one level of cache isn't enough. Typically there are two or three levels of cache, each smaller and faster than the next one out. The **primary (L1) cache** is the fastest cache, usually right on the processor chip and able to serve memory requests in one–three cycles. The **secondary (L2) cache** is larger and slower. Modern processors have at least primary and secondary caches on the processsor chip, often tertiary caches as well. There may be more levels of caching (L3, L4) on separate chip.

For example, the Itanium 2 processor from Intel has three levels of cache right on the chip, with increasing response times (measured in processor cycles) and increasing cache size. The result is that almost all memory requests can be satisfied without going to main memory.

- Primary (L1) cache: 32K, 1 cycle
- Secondary (L2) cache: 96K, 5 cycles
- Tertiary (L3) cache: 3M, 12 cycles
- L4 cache: ~32M, ~25 cycles
- Main memory: off-chip, 100–200 cycles

These numbers are just a rule of thumb, because different processors are configured differently. The Pentium 4 has no on-chip L3 cache; the Core Duo has no L3 cache but has a much larger L2 cache (2MB). Because caches don't affect the instruction set of the processor, architects have a lot of flexibility to change the cache design as a processor family evolves.

Having caches only helps if when the processor needs to get some data, it is already in the cache. Thus, the first time the processor access the memory, it must wait for the data to arrive. On subsequent reads from the same location, there is a good chance that the cache will be able to serve the memory request without involving main memory. Of course, since the cache is much smaller than the main memory, it can't store all of main memory. The cache is constantly throwing out information about memory locations in order to make space for new data. The processor only gets speedup from the cache if the data fetched from memory is still in the cache when it is needed again. When the cache has the data that is needed by the processor, it is called a **cache hit**. If not, it is a **cache miss**. The ratio of the number of hits to misses is called the **cache hit ratio**.

Because memory is so much slower than the processor, the cache hit ratio is critical to overall performance. For example, if a cache miss is a hundred times slower than a cache hit, then a cache miss ratio of 1% means that half the time is spent on cache misses. Of course, the real situation is more complex, because secondary and tertiary caches often handle misses in the primary cache.

The cache records cached memory locations in units of **cache lines** containing multiple words of memory. A typical cache line might contain 4–32 words of memory. On a cache miss, the cache line is filled from main memory. So a series of memory reads to nearby memory locations are likely to mostly hit in the cache. When there is a cache miss, a whole sequence of memory words is requested from main memory at once. This works well because memory chips are designed to make reading a whole series of contiguous locations cheap.

Therefore, caches improve performance when memory accesses exhibit **locality**: accesses are clustered in time and space, so that reads from memory tends to request the same locations repeatedly, or even memory locations near previous requests. Caches are designed to work well with computations that exhibit locality; they will have a high cache hit ratio.

Cache-conscious programming

How does caching affect us as programmers? We would like to write code that has good locality to get the best performance. This has implications for many of the data structures we have looked at, which have varying locality characteristics.

Meta Platforms, Inc.
Exhibit 1059 3/5
Page 003

Arrays

Arrays are implemented as a sequence of consecutive memory locations. Therefore, if the indices used in successive array operations have locality, the corresponding memory addresses will also exhibit locality. There is a big difference between accessing an array sequentially and accessing it randomly, because sequential accesses will produce a lot of cache hits, and random accesses will produce mostly misses. So arrays will give better performance if there is index locality.

Lists

Lists have a lot of pointers. Short lists will fit into cache, but long lists won't. So storing large sets in lists tends to result in a lot of cache misses.

Trees

Trees have a lot of indirections, which can cause cache misses. On the other hand, the top few levels of the tree are likely to fit into the cache, and because they will be visited often, they will tend to be in the cache. Cache misses will tend to happen at the less frequently visited nodes lower in the tree.

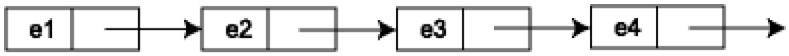
One nice property of trees is that they preserve locality. Keys that are close to each other will tend to share most of the path from the root to their respective nodes. Further, an in-order tree traversal will access tree nodes with good locality. So trees are especially effective when they are used to store sets or maps where accesses to elements/keys exhibit good locality.

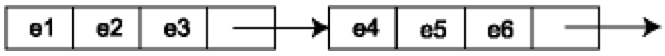
Hash tables

Hash tables are arrays, but a good hash function destroys much locality by design. Accessing the same element repeatedly will bring the appropriate hash bucket into cache. But accesses to keys that are near each other in any ordering on keys will result in memory accesses with no locality. If keys have a lot of locality, a tree data structure may be faster even though it is asymptotically slower!

Chunking

Lists use the cache ineffectively. Accessing a list node causes extra information on the cache line to be pulled into cache, possibly pushing out useful data as well. For best performance, cache lines should be used fully. Often data structures can be tweaked to make this happen. For example, with lists we might put multiple elements into each node. The representation of a bucket set is then a linked list where each node in the linked list contains several elements (and a chaining pointer) and takes up an entire cache line. Thus, we go from a linked list that looks like the one on top to the one on the bottom:





Doing this kind of performance optimization can be tricky in a language like Ocaml where the language is working hard to hide these kind of low-level representation choices from you. A rule of thumb, however, is that OCaml records and tuples are stored nearby in memory. So this kind of memory layout can be implemented to some degree in OCaml, e.g.:

```
type 'a onelist = Null | Cons of 'a * 'a onelist
type 'a chunklist = NullChunk
```

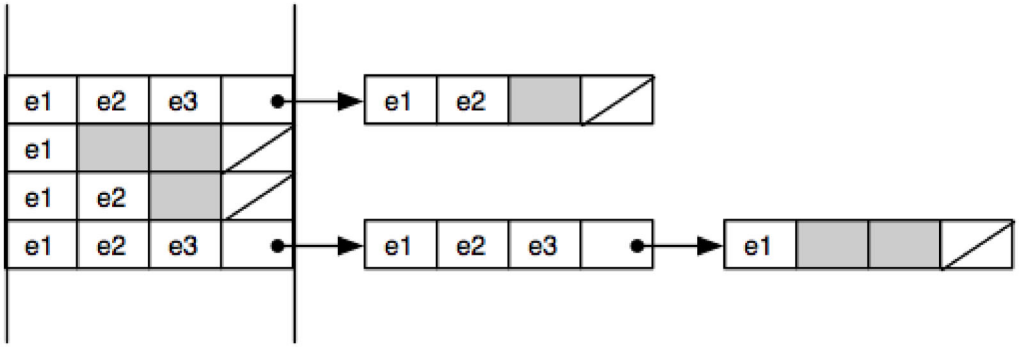

Meta Platforms, Inc.
Exhibit 1059 4/5
Page 004

```
                    | Chunk of 'a * 'a * 'a * 'a * int * 'a chunklist

let cons x (y: 'a chunklist) = match y with
        Chunk(_,_,_,a,1,tail) -> Chunk(x,x,x,a,2,tail)
    |   Chunk(_,_,b,a,2,tail) -> Chunk(x,x,b,a,3,tail)
    |   Chunk(_,c,b,a,3,tail) -> Chunk(x,c,b,a,4,tail)
    |   (Chunk(_,_,_,_,_,_) | NullChunk) -> Chunk(x,x,x,x,1,y)

let rec find (l:int chunklist) (x:int) : bool =
    match l with
        NullChunk -> false
      | Chunk(a,b,c,d,n,t) ->
          (a=x || b=x || c=x || d=x || find t x)
```

The idea is that a chunk stores four elements and a counter that keeps track of the number of element fields (1-4) that are in use. It turns out that scanning down a linked list implemented in this way is about 10 percent faster than scanning down a simple linked list (at least on a 2009 laptop). In a language like C or C++, programmers can control memory layout better and can reap much greater performance benefits by taking locality into account. The ability to carefully control how information is stored in memory is perhaps the best feature of these languages.

Hash tables

Since hash tables use linked lists, they can typically be improved by the same chunking optimization. To avoid chasing the initial pointer from the bucket array to the linked list, we inline some number of entries in each array cell, and make the linked list be chunked as well. This can give substantial speedup but does require some tuning of the various parameters. Ideally each array element should take occupy one cache line on the appropriate cache level. Best performance is obtained at a higher load factor than with a regular linked list. Other pointer-rich data structures can be similarly chunked to improve memory performance.



Meta Platforms, Inc.
Exhibit 1059 5/5
Page 005