Impact of Program and Data Structure on Performance

- Three attributes of the construction of an application can be considered as "structure."
  1. The first of these is the **build structure**, such as how the source code is distributed between the source files.
  2. The second structure is how the source files are combined into applications and supporting **libraries.**
  3. Finally, and probably the most obvious, is that way **data** is organized in the application

Performance and Convenience Trade-Offs in Source Code and Build Structures

- The structure of the source code for an application can cause differences to its performance. Source code is often distributed across source files for the convenience of the developers. It is appropriate that the developers' convenience is one of the main criteria for structuring the sources, but care needs to be taken that it does not cause inconvenience to the user of an application.
- Performance opportunities are lost when the compiler sees only a single file at a time. The single file may not present the compiler with all the opportunities for optimizations that it might have had if it were to see more of the source code. This kind of limitation is visible when a program uses an accessor function—a short function that returns the value of some variable. A trivial optimization is for the compiler to replace this function call with a direct load of the value of the variable.
- There are three common reasons for using static libraries as part of the build process:

1. For "aesthetic" purposes, in that the final linking of the application requires fewer objects. The build process appears to be cleaner because many individual object files are combined into static libraries, and the smaller set appears on the link line. The libraries might also represent bundles of functionality provided to the executable.
2. To produce a similar build process whether the application is built to use static or dynamic libraries. Each library can be provided as either a static or a dynamic version, and it is up to the developer to decide which they will use. This is common when the library is distributed as a product for developers to use.
3. To hide build issues, but this is the least satisfactory reason. For example, an archive library can contain multiple versions of the same routine. At link time, the linker will extract the first version of this routine that it encounters, but it will not warn that there are multiple versions present. If the same code was linked using individual object files without having

first combined the object files into an archive, then the linker would fail to link the executable.

Using Libraries to Structure Applications

Libraries are the usual mechanism for structuring applications as they become larger. There are some **good technical** reasons to use libraries:

- Common functionality can be extracted into a library that can be shared between different projects or applications. This can lead to better code reuse, more efficient use of developer time, and more effective use of memory and disk space.
- Placing functionality into libraries can lead to more convenient upgrades where only the library is upgraded instead of replacing all the executables that use the library.
- Libraries can provide better separation between interface and implementation. The implementation details of the library can be hidden from the users, allowing the implementation of the library to evolve while maintaining a consistent interface.
- Stratifying functionality into libraries according to frequency of use can improve application start-up time and memory footprint by loading only the libraries that are needed. Functionality can be loaded on demand rather than setting up all possible features when the application starts.
- Libraries can be used as a mechanism to dynamically provide enhanced functionality. The functionality can be made available without having to change or even restart the application.
- Libraries can enable functionality to be selected based on the runtime environment or characteristics of the system. For instance, an application may load different optimized libraries depending on the underlying hardware or select libraries at runtime depending on the type of work it is being asked to perform.

There are some **nontechnical** reasons why functionality gets placed into libraries. These reasons may represent the wrong choice for the user.

- Libraries often represent a convenient product for an organizational unit. One group of developers might be responsible for a particular library of code, but that does not automatically imply that a single library represents the best way for that code to be delivered to the end users.
- Libraries are also used to group related functionality. For example, an application might contain a library of string-handling functions. Such a library might be appropriate if it contains a large body of code. On the

other hand, if it contains only a few small routines, it might be more appropriate to combine it with another library.

There are costs associated with libraries. There are a few contributors to cost:

- Library calls may be implemented using a table of function addresses. This table may be a list of addresses for the routines included in a library. A library routine calls into this table, which then jumps to the actual code for the routine.
- Each library and its data are typically placed onto new TLB entries. Calls into a library will usually also result in an ITLB miss and possibly a DTLB miss if the code accesses library-specific data.
- If the library is being lazy loaded (that is, loaded into memory on demand), there will be costs associated with disk access and setting up the addresses of the library functions in memory.
- Unix platforms typically provide libraries as position-independent code. This
  enables the same library to be shared in memory between multiple running applications. The cost of this is an increase in code length. Windows makes the opposite trade-off; it uses position-dependent code in libraries, reducing the opportunity of sharing libraries between running applications but producing slightly faster code.

Rough guidelines for when to use libraries are as follows:

- Libraries make sense when they contain code that is rarely executed. If a substantial amount of code does not need to be loaded from disk for the general use of the application, then the load time of the application can be reduced if this functionality is placed in a library that is loaded only when needed.
- It is useful to place code that is common to multiple applications into shared
  libraries, particularly if the applications are out of the control of the developers of the libraries. This is the situation with most operating systems, where the applications that use the libraries will be developed separately from the core operating system. Most applications use libc, the C standard library, so it makes sense to deliver this library as a shared library that all applications can use. If the internals of the operating system change or if a bug is fixed, then the libraries can be modified without needing to change the applications that use those libraries. This is a form of encapsulation

- Device drivers are usually packaged as libraries. There is a temptation to produce multiple libraries, some of which are core and some of which are device specific. If there is likely to only ever be a single device of a specific type attached to a system, then it is better to provide a single library. If there are likely to be multiple types of devices attached that all share the common core, then it might be appropriate to split the code into device-specific and common code.
- Libraries can also provide dynamically loaded functionality. A library could be released to provide an existing application with new functionality. Placing as much functionality into as few libraries as possible is the most efficient approach, but in many instances the need to dynamically manage functionality will outweigh any overhead induced by packaging the functionality as libraries.

Impact of Data Structures on Performance

- Data structure is probably what most people think of first when they hear the word structure within the context of applications. Data structure is arguably the most critical structure in the program since each data structure will potentially be accessed millions of times during the run of an application. Even a slight gain in performance here can be magnified by the number of accesses and become significant.
- When an application needs an item of data, it fetches it from memory and installs it in cache. The idea with caches is that data that is frequently accessed will become resident in the cache. The cost of fetching data from the cache is substantially lower than the cost of fetching it from memory. Hence, the application will spend less time waiting for frequently accessed data to be retrieved from memory. It is important to realize that each fetch of an item of data will also bring adjacent items into the caches. So, placing data that is likely to be used at nearly the same time in close proximity will mean that when one of the items of data is fetched, the related data is also fetched.
- Often the caches that are closer to the processor have shorter lines, and the lines further from the processor have longer lines. Figure below illustrates what happens when a line is fetched into cache from memory

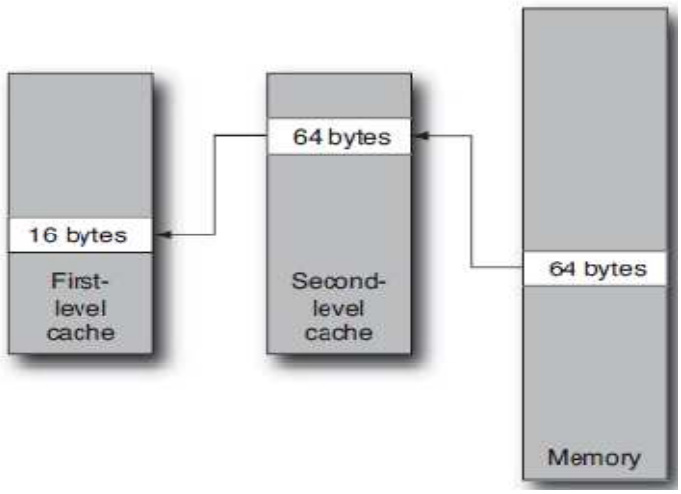

Fig: Fetching data from memory into caches

- On a cache miss, a cache line will be fetched from memory and installed into the second- level cache. The portion of the cache line requested by the memory operation is installed into the first-level cache. In this scenario, accesses to data on the same 16-byte cache line as the original item will also be available from the first-level cache. Accesses to data that share the same 64-byte cache line will be fetched from the second-level cache. Accesses to data outside the 64-byte cache line will result in another fetch from memory.
- If data is fetched from memory when it is needed, the processor will experience the entire latency of the memory operation. On a modern processor, the time taken to perform this fetch can be several hundred cycles. However, there are techniques that reduce this latency:
- ***Out-of-order*** execution is where the processor will search the instruction stream for future instructions that it can execute. If the processor detects a future load instruction, it can fetch the data for this instruction at the same time as fetching data for a previous load instruction. Both loads will be fetched simultaneously, and in the best case, the total cost of the loads can be potentially halved. If more than two loads can be simultaneously fetched, the cost is further reduced.
- ***Hardware prefetching*** of data streams is where part of the processor is dedicated to detecting streams of data being read from memory. When a stream of data is identified, the hardware starts fetching the data before it is requested by the processor. If the hardware prefetch is successful, the data might have become resident in the cache before it was actually needed. Hardware prefetching can be very effective in situations where data is fetched as a stream or through a strided access pattern. It is not able to prefetch data where the access pattern is less apparent.
- ***Software prefetching*** is the act of adding instructions to fetch data from memory before it is needed. Software prefetching has an advantage in

that it does not need to guess where the data will be requested from in the memory, because the prefetch instruction can fetch from exactly the right address, even when the address is not a linear stride from the previous address. Software prefetch is an advantage when the access pattern is nonlinear. When the access pattern is predictable, hardware prefetching may be more efficient because it does not take up any space in the instruction stream.

- Another approach to covering memory latency costs is with CMT rocessors. When one thread stalls because of a cache miss, the other running threads get to use the processor resources of the stalled thread. This approach, unlike those discussed earlier, does not improve the execution speed of a single thread. This can enable the processor to achieve more work by sustaining more active threads, improving throughput rather than single-threaded performance.
- There are a number of common coding styles that can often result in suboptimal layout of data in memory. The following subsections describe each of these.

Improving Performance Through Data Density and Locality

- Paying attention to the order in which variables are declared and laid out in memory can improve performance. As discussed earlier, when a load brings a variable in from memory, it also fetches the rest of the cache line in which the variable resides. Placing variables that are commonly accessed together into a structure so that they reside on the same cache line will lead to performance gains. Consider the structure shown in Listing 1 below

```
struct s
{
int var1;
int padding1[15];
int var2;
int padding2[15];
}
```

- When the structure member var1 is accessed, the fetch will also bring in the surrounding 64 bytes. The size of an integer variable is 4 bytes, so the total size of var1 plus padding1 is 64 bytes. This ensures that the variable var2 is located on the next cache.
- Listing 2 shows the structure reordered so that var1 and var2 are adjacent. This will usually ensure that both are fetched at the same time.

```
struct s
{
int var1;
int var2;
int padding1[15];
```

int padding2[15];
}

- If the structure does not fit exactly into the length of the cache line, there will be situations when the adjacent var1 and var2 are split over two cache lines. Is it better to pack the structures as close as possible to fit as many of them as possible into the same cache line, or is it better to add padding to the structures to make them consistently align with the cache line boundaries? Figure below shows the two situations.

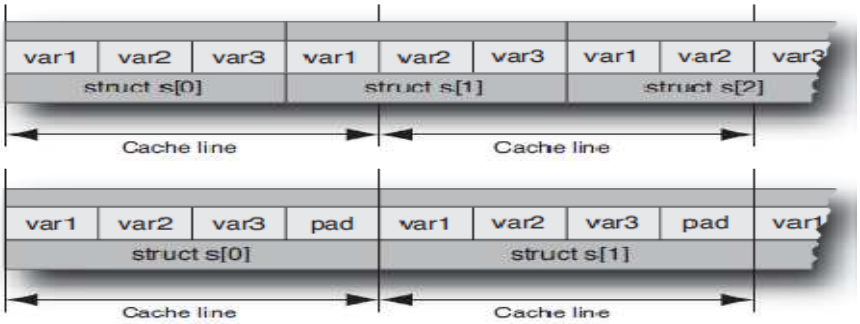

Fig: Using padding to align structures