- NOTE: written in 2007 so consider for design details

Introduction

- computers and components used to be developed together (were simpler)
  - network and memory had comparable speed to CPU
- later, storage and memory subsystem performance fell below CPU performance significantly
  - mostly dealt with using software in OS
  - main memory
  - caches
- removing memory bottleneck occurs in following forms
  - RAM hardware design (speed and parallelism)
  - memory controller designs
  - CPU caches
  - direct memory access (DMA) for devices
- must understand design of commodity hardware to understand memory issues
- (uses Linux as OS context)

Commodity Hardware Today

- horizontal scaling is more common than vertical scaling thus commodity hardware has overtaken specialized software
- standard data center focused around multicore (quad and more) CPUs (with hyper-threading in Intel chips)
  - quad core means 64 VCPUs
- large variance in commodity hardware design
- personal computers and smaller servers

```
      CPU1        CPU2
       V            V
       --------------  FSB (front side bus)
              V
   RAM <- Northbridge
              ^
              V
PCI-E <-> Southbridge <-> SATA
                      <-> USB
```

- all CPUs connected to northbridge via front side bus (FSB)

- northbridge
    - memory controller (contained in northbridge)
        - implementation determines RAM used for machine
        - RAM
        - DRAM
        - Rambus
        - SDRAM
        - all require different memory controllers
- northbridge communicates with all other system devices via southbridge
- southbridge = I/O bridge
    - handles communication with devices via variety of buses
    - common
        - PCI
        - PCI-E (PCI Express)
        - SATA
        - USB
    - less common
        - PATA
        - IEEE 1394
        - serial
        - parallel
    - older systems attached AGP slots to northbridge for performace
- results of structure
    - all communication from one CPU to another must travel over the same bus used to communicate with the northbridge
    - all communication with RAM must pass through the northbridge
    - RAM has only a single port
    - communication between a CPU and a device attached to the southbridge is routed through the northbridge
    - bottlenecks
        - CPU access to RAM
            - can be (partially) resolved with DMA but creates contention for nrothbridge bandwidth
        - bus from northbridge to RAM
        - memory access waits are more prominent with multiple processors, cores, hyper-threads
- memory access patterns greatly influence performance
- some (more expensive) systems allow the northbridge to be attached to many external memory controllers
    - increases bandwidth

        - allows better concurrent access patterns
    - other methods for increasing bandwidth
        - integrate memory controllers into the CPUs and attach memory to each CPU
            - AMD Opteron and SMP systems
            - related approach in Intel Nehalem with CSI (common system interface)
            - results in as many memory banks as CPUs
            - disadvantages
                - results in non-uniform memory access (NUMA)
                - interconnects between CPUs are used when memory attached to another processor is accessed which has a cost
    - commodity NUMA machines are much more common now (2017)

RAM Types

- why both SRAM and DRAM in the same machine?
    - SRAM is faster and provides same functionality
    - SRAM is also much more expensive

Static RAM

- shows structure of 6 transistor SRAM cell
- one cell requires six transistors
- there are variants with four transistors but they have disadvantages
- maintaining the state of the cell requires constant power
- the cell state is available for reading almost immediately once the word access line WL is raised
- the signal is as rectangular (changing quickly between the two binary states) as other transistor-controlled signals
- the cell state is stable, no refresh cycles are needed
- there are other slower and less power hungry SRAM types available

Dynamic RAM

- shows structure of typical DRAM cell here
- keeps state in capacitor and transistor is used to guard access to state
- with a capacitor, reading a cell discharges the capacitor
- the capacitor must be recharged at some point
- takes a short time for the capacity to dissipate, and leakage requires constant refreshes
- no access is possible during refresh phase, which may stall up to 50% of the memory accesses for some workloads

- the tiny charge in a cell requires a sense amplifier to read and distinguish between a 0 and a 1
- reading the cell causes the charge of the capacitor the be depleted so each read must be followed by a recharge
- charging and draining is not instantaneous
- various delays limit how fast DRAM can be
- the chip real estate needed for one DRAM cell is much smaller than for an SRAM cell
- SRAM cells need individual power for the transistors maintaining the state
- the cost wins (except in specialized hardware) which has serious implications for programmer

DRAM Access

- process
  - program selects a memory location using a virtual address
  - processor translates to physical address
  - memory controller selects the RAM chip corresponding to that address
  - to select the individual memory cell on the RAM chip
    - parts of the physical address are passed on in the form of a number of address lines
    - address is passed encoded as a binary number using a smaller set of address lines
    - address passed to the DRAM chip this way must be demultiplexed first
    - a demultiplexer with N address lines will have 2N output lines
    - output lines can be used to select the memory cell
- size of a de-multiplexer increases exponentially with the number of input lines when speed is not to be sacrificed
- using an array the design can get by with one demultiplexer and one multiplexer of half the size, which results in a huge saving
- the row address selection (RAS) demultiplexer select the address lines of a whole row of cells
- when reading the content of all cells is made available to the column address selection (CAS) multiplexer
- content of one column is made available to the data pin of the DRAM chip
  - happens many times in parallel on a number of DRAM chips to produce a total number of bits corresponding to the width of the data bus
- for writing the new cell value is put on the data bus and when the cell is selected using the RAS and CAS it is stored in the cell
- many more complications and need specification for delay

Conclusions

- there are reasons why not all memory is SRAM
- memory cells need to be individually selected to be used
- the number of address lines is directly responsible for the cost of the memory controller, motherboards, DRAM module, and DRAM chip
- it takes a while before the results of the read or write operation are available

DRAM Access Technical Details

Read Access Protocol

Precharge and Activation

Recharging

Memory Types

Conclusions

- accessing DRAM is not fast especially when compared with cache access and CPU speed

Other Main Memory Users

- high-performance cards (network and mass-storage controllers) require DMA
    - can access memory both through northbridge and southbridge
- other buses, like USB, also require FSB bandwidth
    - even if they do not use DMA
- DMA helps but causes contention for FSB bandwidth
- many possibilities for configurations of hardware and memory
- some cheaper systems have graphics systems without separate RAM

CPU Caches

- in 90's speed of CPU cores increased but memory bus and RAM performance did not increase proportionally
- very little very fast RAM loses out to a lot of relatively fast RAM
- computers can have SRAM and DRAM
    - OS would have to optimally distribute data to use SRAM if managing
    - SRAM serves as extension of the register set of the processor

    - SRAM is not managed by OS or user
    - transparently used by processors
- caching takes advantage of temporal and spatial locality (key to purpose/usefulness of caching)
    - for my laptop
        - L1 cache 64 KB per core
        - L2 cache 256 KB per core
        - L3 cache 4 MB to 24 MB shared (8 MB on my mac)
- cache management requires strict replacement and access policies

CPU Caches in the Big Picture

- caches sit before CPU core in path to bus and main memory is on a separate path to bus
    - all loads and stores go through cache
- older systems stuck to von Neumann architecture
- L1 is two separate caches
    - L1i - instruction cache
    - L1d - data cache
- processors have multiple cores and each core can have multiple threads
    - separate cores have separate copies of (almost) all the hardware resources
    - cores can run completely independently unless they are using the same resources
    - threads share almost all of the processor's resources
    - Intel's implementation of threads has only separate registers for the threads and even that is limited, some registers are shared

Cache Operation at High Level

- by default all data read or written by the CPU cores is stored in the cache
    - there are memory regions which cannot be cached but this is something only the OS implementers have to be concerned about
    - there are also instructions which allow the programmer to bypass certain caches
- if the CPU needs a data word the caches are searched first
    - the cache cannot contain the content of the entire main memory
    - each cache entry is tagged using the address of the data word in the main memory
    - a request to read or write to an address can search the caches for a matching tag
    - the address in this context can be either the virtual or physical address (based on the cache implementation)
- it is inefficient to chose a word as the granularity of the cache
- since neighboring memory is likely to be used together it is also be loaded into the cache together

- RAM modules are much more effective if they can transport many data words in a row without a new CAS or even RAS signal
- entries stored in the caches are "lines" of several contiguous words
- older sizes were 32 and 64 bytes
  - if the memory bus is 64 bits wide this means 8 transfers per cache line
- entire cache line is loaded into the L1d
  - the memory address for each cache line is computed by masking the address value according to the cache line size
  - for a 64 byte cache line this means the low 6 bits are zeroed
- when an instruction modifies memory the processor has to load a cache line first because no instruction modifies an entire cache line at once
  - exception: write-combining
  - not possible for a cache to hold partial cache lines
- a cache line which has been written to and which has not been written back to main memory is "dirty"
  - once written the dirty flag is cleared
- an eviction (making more room) from L1d pushes the cache line down into L2
  - room has to be made in L2 which might push the content into L3 and possibly then into main memory
  - each eviction is progressively more expensive
- this is model for exclusive caches on AMD processors
  - Intel implements inclusive caches where each cache line in L1d is also present in L2
  - speeds eviction from L1d
  - disadvantage of wasting memory for content held in two places is minimal
- advantage of exclusive cache is that loading a new cache line only has to touch the L1d and not the L2, which could be faster
- CPUs manage the caches as they like as long as the memory model defined for the processor architecture is adhered to It is, for instance, perfectly fine for a processor to take advantage of little or no memory bus activity and proactively write dirty cache lines back to main memory
- wide variety of cache architectures among the processors for the x86 and x86-64
  - facilitated by memory model abstraction
- in symmetric multi-processor (SMP) systems caches cannot work independently
  - "cache coherency" - all processors should see same memory content at all times
  - processors detect when another processor wants to read or write to a certain cache line rather than having access to the caches of other processors
- if a write access is detected and the processor has a clean copy of the cache, cache line is marked invalid
  - Future references require the cache line to be reloaded
  - read access on another CPU does not necessitate an invalidation
    - multiple clean copies can very well be kept around

- more sophisticated cache implementation
  - cache line is dirty in one processor's cache and a second processor wants to read or write line
  - main memory is out-of-date and the requesting processor must get the cache line content from first processor
  - snooping -> first processor notices this situation and automatically sends the requesting processor the data
    - bypasses main memory
    - in some implementations the memory controller is supposed to notice this direct transfer and store the updated cache line content in main memory
- many cache coherency protocols
  - most important is MESI
    - a dirty cache line is not present in any other processor's cache
    - clean copies of the same cache line can reside in arbitrarily many caches
  - If these rules can be maintained, processors can use their caches efficiently even in multi-processor systems.
- by following rules above, processors in multiprocessor systems need to monitor each others' write accesses and compare the addresses with those in their local caches
  - results in efficient cache use
- TODO: current timing estimates of caches accesses and evictions
- entire cost does not have to be paid for each occurrence of the cache load and miss
  - internal pipelines (of different lengths)
    - instructions are decoded and prepared for execution
    - part of the preparation is loading values from memory (or cache) if they are transferred to a register
    - if the memory load operation can be started early enough in the pipeline it may happen in parallel with other operations
    - the entire cost of the load may be hidden
    - often possible for L1d
    - possible for L2 for some processors with long pipelines
    - if CPU does not have sufficient resources for the memory access or final address of the load becomes available late as the result of another instruction the load costs cannot be hidden (completely)
- the CPU does not necessarily have to wait until the value is safely stored in memory
- for write operations
  - if execution of the following instructions appear to have the same effect as if the value were stored in memory
    - the CPU does not necessarily have to wait until the value is safely stored in memory
    - CPU can take shortcuts and start executing the next instruction early
    - shadow registers -> can hold values no longer available in a regular register

        - with shadow registers it is possible to change the value which is to be stored in the incomplete write operation

CPU Cache Implementation Details

- cache must be able to handle each cell of main memory, with a small ratio of cache to main memory

Associativity

- fully associative cache - each cache line can hold a copy of any memory location
    - processor core would have to compare the tags of each and every cache line with the tag for the requested address to access a cache line
    - tag would be comprised of the entire part of the address which is not the offset into the cache line
- impractical but caches like this exist
- 4MB cache with 64B cache lines the cache would have 65,536 entries
- cache logic would have to be able to pick from all these entries the one matching a given tag in just a few cycles to achieve adequate performance
- a comparator is needed to compare the large tag (note, S is zero) for each line
    - letter next to each connection indicates the width in bits
    - single bit line if none is given
    - ach comparator has to compare two T-bit-wide values
        - appropriate cache line content is selected and made available based on result
        - requires merging as many sets of O data lines as there are cache buckets
    - number of transistors needed to implement a single comparator is large
    - only way to save on the number of comparators is to reduce the number of them by iteratively comparing the tags
- fully associative caches are practical for small caches
    - TLB caches on some Intel processors are fully associative
- different approach is needed for L1i, L1d, and higher level caches
    - restrict the search
    - given the 4MB/64B cache with 65,536 entries we can directly address each entry by using bits 6 to 21 of the address (16 bits)
    - low 6 bits are the index into the cache line
    - direct-mapped cache is fast and relatively easy to implement
    - requires exactly one comparator, one multiplexer and some logic to select only valid cache line content
    - comparator is complex due to the speed requirements but there is only one of them
    - complexity in this approach lies in the multiplexers

        - number of transistors in a simple multiplexer grows with O(log N), where N is the number of cache lines
            - might get slow
            - speed can be increased by spending more real estate on transistors in the multiplexers to parallelize some work
            - total number of transistors can grow slowly with a growing cache size
                - only works well if the addresses used by the program are evenly distributed with respect to the bits used for the direct mapping
                - usually not the case
                - some cache entries are heavily used and repeatedly evicted while
                - others are hardly used at all or remain empty
    - can be solved by making the cache set associative
        - combines the good features of the full associative and direct-mapped caches to largely avoid the weaknesses of those designs
        - tag and data storage are divided into sets, one of which is selected by the address of a cache line
        - similar to the direct-mapped cache
            - instead of only having one element for each set value in the cache a small number of values is cached for the same set value
        - tags for all the set members are compared in parallel, which is similar to the functioning of the fully associative cache
        - not easily overwhelmed by bad or deliberate selection of addresses with the same set numbers
        - size of the cache is not limited by the number of comparators which can be implemented economically
        - if the cache grows it is only the number of columns which increases
        - number of rows (and therefore comparators) only increases if the associativity of the cache is increased
        - processors are using associativity levels of up to 24 for L2 caches or higher
        - L1 caches usually get by with 8 sets.
    - for 4MB/64B cache and 8-way set associativity the cache left has 8,192 sets
        - only 13 bits of the tag are used in addressing the cache set
    - relationship of values for cache size is
        - cache line size -> associativity -> number of sets
    - addresses are mapped into the cache by using
        - O = log2(cache line size)
        - S = log2(number of sets)
    - associativity can help to reduce the number of cache misses significantly
        - for an 8MB cache going from direct mapping to 2-way set associative cache saves almost 44% of the cache misses

    - processor can keep more of the working set in the cache with a set associative cache compared with a direct mapped cache
- in some extreme cases (jump from the 4MB to the 8MB cache) introducing associativity has the same effect as doubling cache size
    - not true for further doubling of the associativity
- in general
    - increasing the associativity of a cache above 8 seems to have little effects for a single-threaded workload
- for hyper-threaded processors
    - first level cache is shared and multi-core processors which use a shared L2 cache
    - basically have two programs hitting on the same cache which causes the associativity in practice to be halved or quartered for quad-core processors
    - increasing numbers of cores the associativity of the shared caches should grow
    - when not possible anymore processor designers have to start using shared L3 caches and beyond, while L2 caches are potentially shared by a subset of the cores
- data for how how the increase in cache size helps with performance cannot be interpreted without knowing about the working set size
    - cache as large as the main memory would lead to better results than a smaller cache
    - no limit to the largest cache size with measurable benefits
    - size of the working set at its peak is 5.6M
    - does not give any absolute number of the maximum beneficial cache size
        - can estimate
    - not all the memory used is contiguous
        - conflicts result even with a 16M cache and a 5.6M working set
        - safe bet that with the same workload the benefits of a 32MB cache would be negligible
- see tests

Measurements of Cache Effects

- to measure program must
    - simulate working sets of arbitrary size
    - read and write access
    - sequential or random access
- sample program creates an array corresponding to the working set size of elements of this type

```
struct l {
    struc l * n;
    long int pad[NPAD];
}
```

- entries are chained in a circular list using the n element, either in sequential or random order
- advancing from one entry to the next always uses the pointer, even if the elements are laid out sequentially
- pad element is the payload and it can grow arbitrarily large
- in some tests the data is modified, in others the program only performs read operations
- working set is made up of an array of struct l elements
- working set of 2N bytes contains elements of num

```
2N / sizeof(struct l)
```

- sizeof(struct l) depends on the value of NPAD
  - 32-bit systems
    - NPAD=7 means the size of each array element is 32 bytes
  - 64-bit systems
    - size is 64 bytes

Single Threaded Sequential Access

- simplest case is a simple walk over all the entries in the list
  - elements are laid out sequentially, densely packed
  - processor can deal with forward and reverse equally well
- measure how long it takes to handle a single list element
- time unit is a processor cycle The first two measurements are polluted by noise. The
- measured workload is simply too small to filter the effects of the rest of the system out
- three distinct levels
  - up to a working set size of 214 bytes
  - from 215 bytes to 220 bytes
  - from 221 bytes and up
- steps
  - processor has a 16kB L1d and 1MB L2
  - no sharp edges in the transition from one level to the other because the caches are used by other parts of the system as well and so the cache is not exclusively available for the program data
  - L2 cache is a unified cache and also used for the instructions (Intel uses inclusive caches)
  - measured times for different working set sizes is unexpected
  - times for the L1d hits are expected
    - load times after an L1d hit are around 4 cycles on the P4

- once the L1d is not sufficient to hold the data one might expect it would take 14 cycles or more per element since this is the access time for the L2
- results show that only about 9 cycles are required
  - discrepancy can be explained by the advanced logic in the processors
  - processor prefetches the next cache line in anticipation of using consecutive memory regions
  - when the next line is actually used it is already halfway loaded
  - delay required to wait for the next cache line to be loaded is therefore much less than the L2 access time

- effect of prefetching is even more visible once the working set size grows beyond the L2 size
- main memory access takes 200+ cycles
- only possible for the processor to keep the access times as low as 9 cycles with effective prefetching is it
- for fewer but larger elements in the list
  - different sizes have the effect that the distance between the n elements in the (still consecutive) list grows
  - four cases
    - 0
    - 56
    - 120
    - 248
- cases all match each other as long as only the L1d is involved
  - no prefetching necessary so all element sizes just hit the L1d for each access
- for the L2 cache hits
  - cases are equal at level of the access time for the L2
  - means prefetching from L2 into L1d is basically disabled
- with NPAD=7 need a new cache line for each iteration of the loop
- for NPAD=0 loop has to iterate eight times before the next cache line is needed
- prefetch logic cannot load a new cache line every cycle
  - stall to load from L2 in every iteration
- when working set size exceeds the L2 capacity see wide variation
  - different element sizes play a big role in the difference in performance
  - processor should recognize the size of the strides and not fetch unnecessary cache lines for NPAD=15 and 31 since the element size is smaller than the prefetch window
  - element size is hampering the prefetching efforts is a result of a limitation of hardware prefetching
    - cannot cross page boundaries
  - reducing the effectiveness of the hardware scheduler by 50% for each size increase
  - if prefetcher were allowed to cross page boundaries and the next page is not resident or valid the OS would have to get involved in locating the page

    - program would experience a page fault it did not initiate itself
        - unacceptable since the processor does not know whether a page is not present or does not exist
    - OS would have to abort the process
    - given that, for NPAD=7 and higher, need one cache line per list element the hardware prefetcher cannot do much
        - no time to load the data from memory since all the processor does is read one word and then load the next element
    - another reason for the slowdown are the misses of the TLB cache
        - cache where the results of the translation of a virtual address to a physical address are stored
        - small since it has to be extremely fast
    - if more pages are accessed repeatedly than the TLB cache has entries for the translation from virtual to physical address has to be constantly repeated
        - costly operation
    - with larger element sizes the cost of a TLB lookup is amortized over fewer elements
        - total number of TLB entries which have to be computed per list element is higher
- describes another test to observe effect of TLB

Single Threaded Random Access

- processor is able to hide most of the main memory and even L2 access latency by prefetching cache lines into L2 and L1d
    - only works well when the memory access is predictable
- if the access pattern is unpredictable or random
    - per-listelement times for the sequential access vs. times when the list elements are randomly distributed in the working set
    - order is determined by the linked list which is randomized
    - no way for the processor to reliably prefetch data
    - can only work by chance if elements which are used shortly after one another are also close to each other in memory
- two important points
    - large number of cycles needed for growing working set sizes
    - machine makes it possible to access the main memory in 200-300 cycles but here we reach 450 cycles and more
        - automatic prefetching is actually working to a disadvantage here
    - can measure the L2 access of the program for the various working set sizes
    - when the working set size is larger than the L2 size, the cache miss ratio (L2 accesses / L2 misses) starts to grow
    - strong correlation between the cycles per list element graph
    - L2 miss rate will grow until it eventually reaches close to 100%

        - given a large enough working set (and RAM) the probability that any of the randomly picked cache lines is in L2 or is in the process of being loaded can be reduced arbitrarily
        - increasing cache miss rate alone explains some of the costs
        - total number of L2 uses per iteration of the program is growing
        - each working set is twice as large as the one before
        - without caching we would expect double the main memory accesses
        - with caches and (almost) perfect predictability the modest increase in the L2 use for sequential access
        - increase is only due to the increase of the working set size
    - for random access the per-element access time more than doubles for each doubling of the working set size
        - average access time per list element increases since the working set size only doubles
            - due to rising rate of TLB misses
        - in the normal case the entire list of randomized as one block
            - other randomizations which are performed in smaller blocks
        - all list elements in the block are traversed before going over to an element in the next block
        - has the effect that number of TLB entries which are used at any one time is limited
    - element size for NPAD=7 is 64 bytes, which corresponds to the cache line size
        - unlikely that the hardware prefetcher has any effect due to the randomized order of the list elements
            - definitely true for anything more than a handful of elements
        - means the L2 cache miss rate does not differ significantly from the randomization of the entire list in one block
        - performance of the test with increasing block size approaches asymptotically the curve for the one-block randomization
            - performance of latter case is significantly influenced by the TLB misses
        - if the TLB misses can be lowered the performance increases significantly

Write Behavior

- caches are supposed to be coherent and this coherency is supposed to be completely transparent for the userlevel code
    - kernel code occasionally requires cache flushes
- if a cache line is modified the result for the system after this point in time is the same as if there were no cache at all and main memory location itself had been modified
- can be implemented by two policies
    - write-through cache
        - simplest way to implement cache coherency
        - if the cache line is written to, the processor immediately also writes the cache line into main memory

        - ensures that, at all times, the main memory and cache are in sync
        - cache content could simply be discarded whenever a cache line is replaced
        - simple but not fast
        - program which modifies a local variable over and over again would create a lot of traffic on the FSB even though the data is likely not used anywhere else
    - write-back cache implementation
        - more complex
        - processor does not immediately write the modified cache line back to main memory
        - cache line is marked as dirty
        - when the cache line is dropped from the cache at some point in the future the dirty bit will instruct the processor to write the data back at that time instead of just discarding the content
        - write-back caches have the chance to be significantly better performing
            - most memory in a system with a decent processor is cached this way
        - processor can take advantage of free capacity on the FSB to store the content of a cache line before the line has to be evacuated
            - allows the dirty bit to be cleared and the processor can just drop the cache line when the room in the cache is needed
        - significant problem
            - when more than one processor (or core or hyper-thread) is available and accessing the same memory it must still be assured that both processors see the same memory content at all times
            - if a cache line is dirty on one processor (i.e., it has not been written back yet) and a second processor tries to read the same memory location, the read operation cannot just go out to the main memory
            - content of the first processor's cache line is needed
- two more cache policies
    - write-combining
        - limited caching optimization more often used for RAM on devices such as graphics cards
        - transfer costs to the devices are much higher than the local RAM access it is even more important to avoid doing too many transfers
        - transferring an entire cache line just because a word in the line has been written is wasteful if the next operation modifies the next word
        - memory for horizontal neighboring pixels on a screen are in most cases neighbors so common occurrence
        - write-combining combines multiple write accesses before the cache line is written out
        - in ideal cases the entire cache line is modified word by word
        - cache line is written to the device only after the last word is written
        - can speed up access to RAM on devices significantly

    - uncacheable
        - usually means the memory location is not backed by RAM at all
        - might be a special address which is hardcoded to have some functionality implemented outside the CPU
        - for commodity hardware this most often is the case for memory mapped address ranges which translate to accesses to cards and devices attached to a bus (PCIe etc)
        - on embedded boards such a memory address can be used to turn an LED on and off
        - caching such an address would obviously be a bad idea
        - LEDs in this context are used for debugging or status reports and one wants to see this as soon as possible
        - memory on PCIe cards can change without the CPU's interaction, so this memory should not be cached
- used for special regions of the address space which are not backed by real RAM
- kernel sets up these policies for the address ranges (on x86 processors using the Memory Type Range Registers, MTRRs)
- rest happens automatically
- MTRRs are also usable to select between write-through and write-back policies

Multi-Processor Support

- multi-core processors have a problem non-shared cache levels (at least the L1d)

    - impractical to provide direct access from one processor to the cache of another processor
    - connection is not fast enough

- practical alternative is to transfer the cache content over to the other processor in case it is needed

- NOTE: also applies to caches which are not shared on the same processor

- perform cache line transfer when one processor needs a cache line which is dirty in another processor's cache for reading or writing

- to determine whether a cache line is dirty in another processor's cache

    - suboptimal to assume a cache line is dirty just because a cache line is loaded by another processor
    - majority of memory accesses are read accesses and the resulting cache lines are not dirty
    - processor operations on cache lines are frequent which means broadcasting information about changed cache lines after each write access is be impractical

- MESI cache coherency protocol (Modified, Exclusive, Shared, Invalid) developed as a result
    - named after the four states a cache line can be in when using the MESI protocol
    - modified
        - local processor has modified the cache line
        - also implies it is the only copy in any cache
    - exclusive
        - cache line is not modified but known to not be loaded into any other processor's cache
    - shared
        - cache line is not modified and might exist in another processor's cache
    - invalid
        - cache line is invalid, i.e., unused
- with these four states it is possible to efficiently implement write-back caches while also supporting concurrent use of read-only data on different processors
- state changes are accomplished little effort (not too much listening between processors)
- certain operations announced on external pins and make the processor's cache handling visible to others
- address of the cache line in question is visible on the address bus
- states and their transitions
    - all cache lines start empty and are set to Invalid
    - if data is loaded into the cache for writing the cache changes to Modified
    - if the data is loaded for reading the new state depends on whether another processor has the cache line loaded as well
    - if this is the case then the new state is Shared, otherwise Exclusive
    - if a Modified cache line is read from or written to on the local processor, the instruction can use the current cache content and the state does not change
    - if a second processor wants to read from the cache line the first processor has to send the content of its cache to the second processor and then it can change the state to Shared
        - data sent to the second processor is also received and processed by the memory controller which stores the content in memory
        - if this did not happen the cache line could not be marked as Shared
    - if the second processor wants to write to the cache line the first processor sends the cache line content and marks the cache line locally as Invalid
        - "Request For Ownership" (RFO) operation
        - performing this operation in the last level cache is expensive

        - for write-through caches also have to add the time it takes to write the new cache line content to the next higher-level cache or the main memory, further increasing the cost
    - if a cache line is in the Shared state and the local processor reads from it no state change is necessary and the read request can be fulfilled from the cache
    - if the cache line is locally written to
        - cache line can be used as well but the state changes to Modified
        - requires that all other possible copies of the cache line in other processors are marked as Invalid
        - write operation has to be announced to the other processors via an RFO message
    - if the cache line is requested for reading by a second processor nothing has to happen
        - main memory contains the current data and the local state is already Shared
    - in case a second processor wants to write to the cache line (RFO) the cache line is simply marked Invalid
        - no bus operation is needed

- Exclusive state is mostly identical to the Shared state with one crucial difference
    - a local write operation does not have to be announced on the bus
    - local cache is known to be the only one holding this specific cache line
    - can be an advantage so the processor will try to keep as many cache lines as possible in the Exclusive state instead of the Shared state
    - Shared is the fallback in case the information is not available at that moment
    - Exclusive state can also be left out completely without causing problems
        - only the performance that will suffer since the E!M transition is much faster than the S!M transition

- filling caches is expensive but now we also have to look out for

- RFO messages are expensive
    - a thread is migrated from one processor to another and all the cache lines have to be moved over to the new processor
    - acache line is actually needed in two different processors

- multi-thread or multi-process programs always need some synchronization
    - implemented using memory
    - there are some necessary RFO messages
    - have to be kept as infrequent as possible

- cache coherency protocol messages must be distributed among the processors of the system
    - MESI transition cannot happen until it is clear that all the processors in the system have had a chance to reply to the message
    - longest possible time a reply can take determines the speed of the coherency protocol
        - collisions on the bus are possible
        - latency can be high in NUMA systems
        - traffic volume can slow things down
    - avoiding unnecessary traffic!
- FSB is a shared resource and thus always causes a problem that is machine specific on multiprocessor machines
    - (most machines) all processors are connected via one single bus to the memory controller
    - if a single processor can saturate the bus (as is usually the case) then two or four processors sharing the same bus will restrict the bandwidth available to each processor even more
    - even if each processor has its own bus to the memory controller there is still the bus to the memory modules
    - even in the extended model concurrent accesses to the same memory module will limit the bandwidth
    - also true with the AMD model where each processor can have local memory
        - all processors can concurrently access their local memory quickly
        - especially with the integrated memory controller
        - multi-thread and multi-process programs at least occasionally have to access the same memory regions to synchronize
- concurrency is severely limited by the finite bandwidth available for the implementation of the necessary synchronization
    - programs need to be carefully designed to minimize accesses from different processors and cores to the same memory locations

Multi Threaded Access

- performance graphs of concurrently using the same cache lines on different processors
    - more than one thread is running at the same time
- measured fastest runtime of any of the threads
    - means the time for a complete run when all threads are done is even higher
    - four processors
    - tests use up to four threads

    - all processors share one bus to the memory controller and there is only one bus to the memory modules
- performance for sequential read-only access for 128 bytes entries (NPAD=15 on 64-bit machines)
    - behavior when running multiple threads
    - NOTE: no memory is modified and no attempts are made to keep the threads in sync when walking the linked list
    - no RFO messages are necessary
    - all the cache lines can be shared
    - test results show 18% performance decrease for the fastest thread when two threads are used and up to 34% when four threads are used
    - since no cache lines have to be transported between the processors
    - slow-down is solely caused by the one or both of the two bottlenecks
        - shared bus from the processor to the memory controller
        - bus from the memory controller to the memory modules
    - when working set size is larger than the L3 cache in this machine all three threads will prefetch new list elements
    - even with two threads the available bandwidth is not sufficient to scale linearly (i.e., no penalty from running multiple threads)
- results for the sequential Increment test
    - about a 18% penalty for running two threads and now an amazing 93% penalty for running four threads
        - means the prefetch traffic together with the write-back traffic is pretty much saturating the bus when four threads are used
    - as soon as more than one thread is running, the L1d is basically ineffective
    - single-thread access times exceed 20 cycles only when the L1d is not sufficient to hold the working set
    - when multiple threads are running
        - access times are hit immediately
        - even with the smallest working set sizes
- hard to measure with test program
    - test modifies memory and must expect RFO messages
    - do not see higher costs for the L2 range when more than one thread is used
    - program would have to use a large amount of memory and all threads must access the same memory in parallel . This is hard to achieve without a lot of synchronization which would then dominate the execution time.
- Add next last test with random access of memory
    - takes around 1,500 cycles to process a single list element in the extreme case
    - best possible speed-up the test program incurs for the largest working set size by using two or four threads
    - for two threads the theoretical limits for the speed-up are 2 and, for four threads, 4

    - numbers for two threads are not bad
    - for four threads the numbers for the last test show that it mostly not worth it to scale beyond two threads
        - have to ignore the smallest sizes
        - for the range of the L2 and L3 cache we can see that we indeed achieve almost linear acceleration
            - almost reach factors of 2 and 4 respectively
        - as soon as the L3 cache is not sufficient to hold the working set the numbers drop drastically
    - speed-up of two and four threads is identical
- one of the reason why it is difficult to find motherboards with sockets for more than four CPUs all using the same memory controller
    - machines with more processors have to be built differently
- working sets which fit into the last level cache will not allow linear speed-ups in some cases
    - the norm since threads are usually not very decoupled
- possible to work with large working sets and still take advantage of more than two threads . Doing this requires thought, though. We will talk about some approaches in section 6.

Special Case: Hyper-Threads

- hyper-Threads (sometimes called Symmetric Multi-Threading, SMT) are implemented by the CPU
- individual threads cannot really run concurrently
- all share almost all the processing resources except for the register set
- individual cores and CPUs still work in parallel but the threads implemented on each core are limited by this restriction
- in theory there can be many threads per core
    - so far Intel's CPUs at most have two threads per core
- CPU is responsible for time-multiplexing the threads
- real advantage is that the CPU can schedule another hyper-thread and take advantage of available resources such as arithmetic logic units (ALUs) when the currently running hyper-thread is delayed
    - in most cases this is a delay caused by memory accesses
- if two threads are running on one hyper-threaded core the program is only more efficient than the single-threaded code if the combined runtime of both threads is lower than the runtime of the single-threaded code
    - possible by overlapping the wait times for different memory accesses which usually would happen sequentially
- simple calculation shows the minimum requirement on the cache hit rate to achieve a certain speed-up
- execution time for a program can be approximated with a simple model with only one level of cache

```
Texe = N[(1 − Fmem)Tproc + Fmem(GhitTcache + (1 − Ghit)Tmiss)]
```

- for it to make any sense to use two threads the execution time of each of the two threads must be at most half of that of the single-threaded code
  - only variable on either side is the number of cache hits
- see graph for minimum cache hit rate required to not slow down the thread execution by 50%
- hyper-threads are therefore only useful in a limited range of situations
  - cache hit rate of the single-threaded code must be low enough that given the equations above and reduced cache size the new hit rate still meets the goal
  - only then can it make any sense to use hyper-threads
  - result is faster in practice depends on whether the processor is sufficiently able to overlap the wait times in one thread with execution times in the other threads
  - overhead of parallelizing the code must be added to the new total runtime and this additional cost often cannot be neglected
- if the two hyper-threads execute completely different code (i.e., the two threads are treated like separate processors by the OS to execute separate processes) the cache size is indeed cut in half
  - means a significant increase in cache misses
  - OS scheduling practices like this are questionable unless the caches are sufficiently large
- might be best to turn off hyper-threads in the computer's BIOS unless the workload for the machine consists of processes which can definitely benefit from hyper-threads (TODO: check on this now)

Other Details

- all relevant processors today provide virtual address spaces to processes
- two different kinds of addresses
  - virtual
  - physical
- virtual addresses is that they are not unique
  - can over time refer to different physical memory addresses
  - same address in different processes also likely refers to different physical addresses
- virtual addresses used during execution which must to be translated with the help of the Memory Management Unit (MMU) into physical addresses
  - non-trivial
- in the pipeline to execute an instruction the physical address might only be available at a later stage
  - means that the cache logic has to be very quick in determining whether the memory location is cached

- if virtual addresses could be used the cache lookup can happen much earlier in the pipeline and in case of a cache hit the memory content can be made available
    - result is that more of the memory access costs could be hidden by the pipeline
- designers are currently using virtual address tagging for the first level caches
    - caches are small and can be cleared easily
    - at least partial clearing the cache is necessary if the page table tree of a process changes
    - might be possible to avoid a complete flush if the processor has an instruction which allows to specify the virtual address range which has changed
- using virtual addresses is almost mandatory because of the low latency of L1i and L1d caches (3 cycles)
- physical address tagging is needed for larger caches (L2,L3,...)
    - caches have a higher latency and the virtual!physical address translation can finish in time
    - flushing thes caches often would be costly
        - larger (i.e., a lot of information is lost when they are flushed)
        - refilling them takes a long time due to the main memory access latency
- in general not necessary to know about the details of the address handling in those caches
    - cannot be changed and all the factors which would influence the performance are normally something which should be avoided or is associated with high cost
- overflowing the cache capacity is bad and all caches run into problems early if the majority of the used cache lines fall into the same set
    - latter can be avoided with virtually addressed caches but is impossible for user-level processes to avoid for caches addressed using physical addresses
- REMEMBER: don't map the same physical memory location to two or more virtual addresses in the same process if at all possible
- cache replacement strategy
    - most caches evict the Least Recently Used (LRU) element first
    - always a good default strategy
    - with larger associativity maintaining the LRU list becomes more and more expensive
    - much a programmer can do
    - if the cache is using physical address tags there is no way to find out how the virtual addresses correlate with the cache sets
    - it is the job of the OS to arrange that this does not happen too often
- with virtualization things get even more complicated
    - not even the OS has control over the assignment of physical memory
    - Virtual Machine Monitor (VMM, aka Hypervisor) is responsible for the physical memory assignment
- for programmer
    - use logical memory pages completely

    - use page sizes as large as meaningful to diversify the physical addresses as much as possible
        - larger page sizes have other benefits

Instruction Cache

- instructions executed by the processor are also cached
    - this cache is much less problematic than the data cache
    - quantity of code which is executed depends on the size of the code that is needed
        - size of the code in general depends on the complexity of the problem which is fixed
    - program's data handling is designed by the programmer the program's instructions are usually generated by a compiler
        - compiler writers know about the rules for good code generation
    - program flow is much more predictable than data access patterns
        - CPUs are very good at detecting patterns which helps with prefetching
    - code always has quite good spatial and temporal locality
- CPUs have been designed with pipelines since the core clock of CPUs increased and the difference in speed between cache and core grew
- means the execution of an instruction happens in stages
    - instruction is decoded
    - parameters are prepared
    - finally it is executed
- such a pipeline can be long (> 20 stages for Intel's Netburst architecture)
    - long pipeline means that if the pipeline stalls (i.e., the instruction flow through it is interrupted) it takes a while to get up to speed again
    - e.g. happens if the location of the next instruction cannot be correctly predicted or if it takes too long to load the next instruction (e.g., when it has to be read from memory)
- CPU designers spend a lot of time and chip real estate on branch prediction so that pipeline stalls happen as infrequently as possible
- on CISC processors the decoding stage can also take some time
    - x86 and x86-64 processors are especially affected
        - processors therefore do not cache the raw byte sequence of the instructions in L1i but instead they cache the decoded instructions
    - L1i in this case is called the "trace cache"
    - trace caching allows the processor to skip over the first steps of the pipeline in case of a cache hit which is especially good if the pipeline stalled
- caches from L2 on are unified caches which contain both code and data
    - code is cached in the byte sequence form and not decoded

- rules related to the instruction cache for best performance
    - generate code which is as small as possible
        - exceptions
            - when software pipelining for the sake of using pipelines requires creating more code
            - where the overhead of using small code is too high.
    - help the processor to make good prefetching decisions
        - can be done through code layout or with explicit prefetching
    - rules are usually enforced by the code generation of a compiler

3.4.1 Self Modifying Code

- in past memory was contended over
    - lots of effort to reduce the size of a program to make more room for program data
    - one trick was to change the program itself over time
- Self Modifying Code (SMC)
    - for performance reasons
    - security exploits
- SMC should in general be avoided
- code which is changed cannot be kept in the trace cache which contains the decoded instructions
    - if an upcoming instruction is changed while it already entered the pipeline the processor has to throw away a lot of work and start all over again
- since the processor assumes–for simplicity reasons and because it is true in 99.9999999% of all cases that the code pages are immutable, the L1i implementation does not use the MESI protocol but instead a simplified SI protocol
    - if modifications are detected a lot of pessimistic assumptions have to be made It is highly advised to avoid SMC whenever possible. Memory is not such a scarce resource anymore. It is
- better to write separate functions instead of modifying one function according to specific needs
- if SMC absolutely has to be used
    - write operations should bypass the cache as to not create problems with data in L1d needed in L1i
- easy to recognize on Linux
    - all program code is write-protected when built with the regular toolchain
    - programmer has to perform significant magic at link time to create an executable where the code pages are writable
    - modern Intel x86 and x86-64 processors have dedicated performance counters which count uses of self-modifying code
    - possible to recognize programs with SMC using these counters (even if the program will succeed due to relaxed permissions)

3.5 Cache Miss Factors

- costs increase drastically with cache misses
- sometimes unavoidable
    - important to understand the actual costs and what can be done to mitigate the problem

3.5.1 Cache and Memory Bandwidth

- measure the bandwidth available in optimal circumstances to get a better understanding of the capabilities of processors
    - processor versions vary widely
- program to measure performance uses the SSE instructions of the x86 and x86-64 processors to load or store 16 bytes at once
    - working set is increased from 1kB to 512MB just as in our other tests and it is measured how many bytes per cycle can be loaded or stored
- 64-bit Intel Netburst processor (see charts)
    - for working set sizes which fit into L1d the processor is able to read the full 16 bytes per cycle
        - one load instruction is performed per cycle (the movaps instruction moves 16 bytes at once)
    - as soon as the L1d is not sufficient anymore the performance goes down dramatically to less than 6 bytes per cycle
    - there is a step at 218 bytes which is due to the exhaustion of the DTLB cache
    - reading is sequential so prefetching can predict the accesses and the FSB can stream the memory content
    - prefetched data is not propagated into L1d
    - resulting numbers are not achievable in a real program and should be though of as practical limits
- write performance, even for small working set sizes, does not ever rise above 4 bytes per cycle
    - indicates that in these Netburst processors, Intel elected to use a Write-Through mode for L1d where the performance is obviously limited by the L2 speed
    - also means that the performance of the copy test, which copies from one memory region into a second non-overlapping memory region, is not significantly worse
- for write and copy -> low performance once the L2 cache is not sufficient anymore
    - means write operations are by a factor of ten slower than the read operations
    - optimizing those operations is even more important for the performance of the program
- same processor but with two threads running, one pinned to each of the two hyper-threads of the processor
    - hyper-threads share all the resources except the registers each thread has only half the cache and bandwidth available

  - means even though each thread has to wait a lot and could award the other thread with execution time this does not make any difference since the other thread also has to wait for the memory
    - shows the worst possible use of hyper-threads.
- Intel Core 2 processor
  - dual-core processor with shared L2 which is four times as big as the L2 on the P4 machine
    - accounts for delayed drop-off of the write and copy performance
  - read performance throug the working set range stays close to optimal
  - drop-off in the read performance after 220 bytes is due to the working set being too big for the DTLB
    - high numbers means the processor
      - able to prefetch the data
      - transport the data in time
      - data is prefetched into L1d
  - processor does not have a Write-Through policy
    - written data is stored in L1d and only evicted when necessary
    - allows for write speeds close to optimal
  - once L1d is not sufficient anymore the performance drops significantly
  - as with the Netburst the write performance is much lower
    - when even the L2 is not sufficient anymore the speed difference increases to a factor of 20!
  - Core 2 processors do not perform poorly
    - performance is always better than the Netburst core's
- two threads, one on each of the two cores of the Core 2 processor
  - both threads access the same memory, not necessarily perfectly in sync
  - results for the read performance are not different from the single-threaded case
  - write and copy performance for working set sizes which would fit into L1d
    - performance is the same as if the data had to be read from the main memory
    - both threads compete for the same memory location and RFO messages for the cache lines have to be sent
  - problem is that these requests are not handled at the speed of the L2 cache, even though both cores share the cache
    - when L1d cache is not sufficient anymore modified entries are flushed from each core's L1d into the shared L2
    - at that point the performance increases significantly since now the L1d misses are satisfied by the L2 cache and RFO messages are only needed when the data has not yet been flushed
    - this is why there is a 50% reduction in speed for these sizes of the working set

    - since both cores share the same FSB each core gets half the FSB bandwidth which means for large working sets each thread's performance is about half that of the single threaded case
- AMD family 10h Opteron processor
    - see source doc for details

3.5.2 Critical Word Load

- memory is transferred from the main memory into the caches in blocks which are smaller than the cache line size
    - 64 bits are transferred at once and the cache line size is 64 or 128 bytes
    - means 8 or 16 transfers per cache line are needed
- DRAM chips can transfer those 64-byte blocks in burst mode
    - can fill the cache line without any further commands from the memory controller and the possibly associated delays
- if the processor prefetches cache lines probably the best way to operate
- if a program's cache access of the data or instruction caches misses
    - compulsory cache miss because the data is used for the first time
    - capacity cache miss because the limited cache size requires eviction of the cache line
    - word inside the cache line which is required for the program to continue might not be the first word in the cache line
    - even in burst mode and with double data rate transfer the individual 64-bit blocks arrive at noticeably different times
    - each block arrives 4 CPU cycles or more later than the previous one
    - if the word the program needs to continue is the eighth of the cache line the program has to wait an additional 30 cycles or more after the first word arrives
- memory controller is free to request the words of the cache line in a different order
- critical word - the word the program is currently waiting on
- processor can communicate the critical word and the memory controller can request this word first
    - once the word arrives the program can continue while the rest of the cache line arrives and the cache is not yet in a consistent state
    - technique is called Critical Word First & Early Restart
- modern processors implement this technique
- situations when not possible
    - if the processor prefetches data the critical word is not known
    - if the processor requests the cache line during the time the prefetch operation is in flight it will have to wait until the critical word arrives without being able to influence the order
    - hyper-threads, by definition share everything but the register set
        - includes the L1 caches

    - for individual cores of a processor
        - each core has at least its own L1 caches
        - early multi-core processors had no shared caches at all
        - later Intel models have shared L2 caches for dual-core processors
            - for quad-core processors have to deal with separate L2 caches for each pair of two cores
                - no higher level caches
        - AMD's family 10h processors have separate L2 caches and a unified L3 cache
- having no shared cache has an advantage if the working sets handled by the cores do not overlap
    - works well for single-threaded programs
    - always some overlap
        - caches all contain the most actively used parts of the common runtime libraries which means some cache space is wasted
- completely sharing all caches beside L1 as Intel's dual-core processors can have advantage
    - if the working set of the threads working on the two cores overlaps significantly the total available cache memory is increased and working sets can be bigger without performance degradation
    - if the working sets do not overlap Intel's Advanced Smart Cache management is supposed to prevent any one core from monopolizing the entire cache
- if both cores use about half the cache for their respective working sets there is conflict
    - cache constantly has to weigh the two cores' cache use and the evictions performed as part of this rebalancing might be chosen poorly
    - see test program and results
- position of the critical word on a cache line matters

3.5.3 Cache Placement

- cache placement in relationship to the hyperthreads, cores, and processors is not under control of the programmer
    - programmers can determine where the threads are executed and then it becomes important how the caches relate to the used CPUs
- only describe architecture details which the programmer has to take into account when setting the affinity of the threads
- know the workloads and the details of the machine architecture to achieve the best performance

3.5.4 FSB Influence

- plays a central role in the performance of the machine

    - cache content can only be stored and loaded as quickly as the connection to the memory allows

  - can demonstrate by running a program on two machines which only differ in the speed of their memory modules
  - numbers show that there is a large benefit when the FSB is really stressed for large working set size
  - maximum performance increase measured in the test is close to the theoretical maximum
  - shows that a faster FSB indeed can pay off big time
    - not critical when the working set fits into the caches
  - working set of a system comprises the memory needed by all concurrently running processes
    - easily possible to exceed 4MB memory or more with much smaller programs. Today some of Intel's processors support FSB speeds up to 1,333MHz which would mean another 60% increase. The future is going to see even higher speeds.

- fast RAM and high FSB speeds are worth the cost if speed is important and the working set sizes are large

- even though the processor might support higher FSB speeds the motherboard/Northbridge might not

  - check the specifications

4 Virtual Memory

- virtual memory (VM) subsystem of a processor implements the virtual address spaces provided to each process
  - makes each process think it is alone in the system
- concentrate on the actual implementation details of the virtual memory subsystem and the associated costs
- virtual address space is implemented by the Memory Management Unit (MMU) of the CPU
  - OS has to fill out the page table data structures, but most CPUs do the rest of the work themselves
  - complicated mechanism
  - best way to understand by looking at the data structures used to describe the virtual address space
- input to the address translation performed by the MMU is a virtual address
  - there are usually few–if any restrictions on its value
- virtual addresses are 32-bit values on 32-bit systems, and 64-bit values on 64-bit systems
- on some systems, for instance x86 and x86-64, the addresses used actually involve another level of indirection
  - use segments which simply cause an offset to be added to every logical address
  - can ignore this part of address generation

4.1 Simplest Address Translation

- translation of the virtual address to a physical address
    - MMU can remap addresses on a page-by-page basis
    - virtual address is split into distinct parts (like cache line addressing)
    - parts are used to index into various tables which are used in the construction of the final physical address
- for the simplest model there is only one level of tables

```
Virtual Address
|  Directory  |    Offset    |    ->  Physical Page
       |                              physical address
       V
Page directory                     /
directory entry                 --
```

Figure 4.1 shows how the different parts of the virtual address are used. A

- top part is used to select an entry in a Page Directory
    - each entry in that directory can be individually set by the OS
    - page directory entry determines the address of a physical memory page
        - more than one entry in the page directory can point to the same physical address
    - complete physical address of the memory cell is determined by combining the page address from the page directory with the low bits from the virtual address
    - page directory entry also contains some additional information about the page such as access permissions
    - data structure for the page directory is stored in main memory
    - OS has to allocate contiguous physical memory and store the base address of this memory region in a special register
    - appropriate bits of the virtual address are then used as an index into the page directory, which is actually an array of directory entries
- layout used for 4MB pages on x86 machines
    - Offset part of the virtual address is 22 bits in size, enough to address every byte in a 4MB page
    - remaining 10 bits of the virtual address select one of the 1024 entries in the page directory
    - entry contains a 10 bit base address of a 4MB page which is combined with the offset to form a complete 32 bit address

4.2 Multi-Level Page Tables

- 4MB pages are not the norm
    - waste a lot of memory since many operations an OS has to perform require alignment to memory pages
- with 4kB pages (the norm on 32-bit machines and, still, often on 64-bit machines), the Offset part of the virtual address is only 12 bits in size
    - leaves 20 bits as the selector of the page directory
- a table with 220 entries is not practical
    - if each entry would be only 4 bytes the table would be 4MB in size
    - with each process potentially having its own distinct page directory much of the physical memory of the system would be tied up for these page directories
- use multiple levels of page tables
    - levels form a huge, sparse page directory
        - address space regions which are not actually used do not require allocated memory
    - representation is much more compact, making it possible to have the page tables for many processes in memory without impacting performance too much
- most complicated page table structures comprise four levels

```
Virtual Address
| Level 4 Index | Level 3 Index | Level 2 Index | Level 1 Index | Offset |
       |                |               |               |            |
Level 4 Directory       V               V               V            V
level 4 entry
            ->   Level 3 Directory
                 level 3 entry
                             ->  Level 2 Directory
                                 level 2 entry
                                              -> Level 1 Directory
                                                 level 1 entry
                                                             -> Physical Page
                                                                physical address
```

- virtual address is split into at least five parts
    - four are indexes into the various directories
    - level 4 directory is referenced using a special-purpose register in the CPU
    - content of the level 4 to level 2 directories is a reference to next lower level directory
    - if a directory entry is marked empty it obviously need not point to any lower directory
        - page table tree can be sparse and compact
    - entries of the level 1 directory are partial physical addresses, plus auxiliary data like access permissio

- processor first determines the address of the highest level directory to determine the physical address corresponding to a virtual address
    - address is usually stored in a register
    - CPU takes the index part of the virtual address corresponding to this directory and uses that index to pick the appropriate entry
    - entry is the address of the next directory, which is indexed using the next part of the virtual address
    - process continues until it reaches the level 1 directory, at which point the value of the directory entry is the high part of the physical address
    - physical address is completed by adding the page offset bits from the virtual address
    - process is called page tree walking
    - some processors (like x86 and x86-64) perform this operation in hardware, others need assistance from the OS
- each process running on the system might need its own page table tree
    - possible to partially share trees but this is rather the exception
    - good for performance and scalability if the memory needed by the page table trees is as small as possible
    - ideal case for this is to place the used memory close together in the virtual address space; the actual physical addresses used do not matter.
- a small program might get by with using just one directory at each of levels 2, 3, and 4 and a few level 1 directories
    - on x86-64 with 4kB pages and 512 entries per directory this allows the addressing of 2MB with a total of 4 directories (one for each level)
    - 1GB of contiguous memory can be addressed with one directory for levels 2 to 4 and 512 directories for level 1
- assuming all memory can be allocated contiguously is too simplistic
    - stack and the heap area of a process are, in most cases, allocated at pretty much opposite ends of the address space
    - allows either area to grow as much as possible if needed
    - means that there are most likely two level 2 directories needed and correspondingly more lower level directories
    - does not always match current practice
- for security reasons the various parts of an executable (code, data, heap, stack, Dynamic Shared Objects (DSOs), aka shared libraries) are mapped at randomized addresses
    - randomization extends to the relative position of the various parts
        - implies that the various memory regions in use in a process are widespread throughout the virtual address space
    - by applying some limits to the number of bits of the address which are randomized the range can be restricted in most cases will not allow a process to run with just one or two directories for levels 2 and 3

    - if performance is really much more important than security, randomization can be turned off
    - OS will then usually at least load all DSOs contiguously in virtual memory

4.3 Optimizing Page Table Access

- all the data structures for the page tables are kept in the main memory
    - where the OS constructs and updates the tables
    - on creation of a process or a change of a page table the CPU is notified
- page tables are used to resolve every virtual address into a physical address using the page table walk
- at least one directory for each level is used in the process of resolving a virtual address
    - requires up to four memory accesses (for a single access by the running process)
    - this is slow
    - it is possible to treat these directory table entries as normal data and cache them in L1d, L2, etc.
    - also slow
- CPU designers have used a different optimization for a long time
    - simple computation can show that only keeping the directory table entries in the L1d and higher cache leads to bad performance
    - each absolute address computation would require a number of L1d accesses corresponding to the page table depth
    - accesses cannot be parallelized since they depend on the previous lookup's result
    - on a machine with four page table levels this would require at the very least 12 cycles
    - adds to that the probability of an L1d miss and the result is nothing the instruction pipeline can hide
    - additional L1d accesses also steal precious bandwidth to the cache
- instead of just caching the directory table entries the complete computation of the address of the physical page is cached
    - since the page offset part of the virtual address does not play any part in the computation of the physical page address, only the rest of the virtual address is used as the tag for the cache
    - depending on the page size this means hundreds or thousands of instructions or data objects share the same tag and therefore same physical address prefix
- cache into which the computed values are stored is called the Translation Look-Aside Buffer (TLB)
    - usually a small cache since it has to be extremely fast
    - modern CPUs provide multi-level TLB caches
        - higher-level caches are larger and slower
        - small size of the L1TLB is often made up for by making the cache fully associative, with an LRU eviction policy

        - this cache has been growing in size and, in the process, was changed to be set associative
            - might not be the oldest entry which gets evicted and replaced whenever a new entry has to be added
- tag used to access the TLB is a part of the virtual address
    - if the tag has a match in the cache, the final physical address is computed by adding the page offset from the virtual address to the cached value
    - fast process
    - has to be since the physical address must be available for every instruction using absolute addresses and for L2 look-ups which use the physical address as the key
    - can be quite costly if the TLB lookup misses the processor has to perform a page table walk
- prefetching code or data through software or hardware could implicitly prefetch entries for the TLB if the address is on another page
    - cannot be allowed for hardware prefetching because the hardware could initiate page table walks that are invalid
    - programmers cannot rely on hardware prefetching to prefetch TLB entries
    - it has to be done explicitly using prefetch instructions
    - TLBs can appear in multiple levels
    - two types of TLB
        - an instruction TLB (ITLB)
        - data TLB (DTLB)
    - higher-level TLBs such as the L2TLB are usually unified, as is the case with the other caches

4.3.1 Caveats Of Using A TLB

- TLB is a global resource for processor cores
    - all threads and processes executed on the processor core use the same TLB
    - CPU cannot blindly reuse the cached entries if the page table is changed since the translation of virtual to physical addresses depends on which page table tree is installed
    - each process has a different page table tree as does the kernel and the VMM (hypervisor) if present
        - threads in the same process don't
- can handle problem that it is also possible that the address space layout of a process changes by
    - TLB is flushed whenever the page table tree is changed
        - TLB is flushed whenever a context switch is performed

- a switch from one thread/process to another requires executing some kernel code in most OSes
    - TLB flushes are restricted to leaving (and sometimes entering) the kernel address space
- also happens on virtualized systems when the kernel has to call the VMM and on the way back if
    - kernel and/or VMM does not have to use virtual addresses
    - can reuse the same virtual addresses as the process or kernel which made the system/VMM call (i.e. address spaces are overlaid)
- TLB only has to be flushed if the processor resumes execution of a different process or kernel upon leaving the kernel or VMM
- flushing the TLB is effective but expensive
- kernel code might be restricted to a few thousand instructions which when executing a system call touch a handful of new pages (or one huge page, as is the case for Linux on some architectures)
    - would replace only as many TLB entries as pages are touched
- full flush would mean that more than 100 and 200 entries (respectively) would be flushed unnecessarily on Intel's Core2 architecture with its 128 ITLB and 256 DTLB entries
- all flushed TLB entries can be used again but they will be gone when the system call returns to the same process
    - same is true for often-used code in the kernel or VMM
- on each entry into the kernel the TLB has to be filled even though the page tables for the kernel and VMM usually do not change
    - in theory TLB entries could be preserved for a long time
    - explains why the TLB caches in today's processors are not bigger
        - programs most likely will not run long enough to fill all entries
- one possibility to optimize the cache flushes is to individually invalidate TLB entries
    - e.g. if the kernel code and data falls into a specific address range only the pages falling into this address range have to evicted from the TLB
    - only requires comparing tags and is not very expensive
    - also useful in case a part of the address space is changed through a call to munmap

- tags for the TLB entries are extended to additionally and uniquely identify the page table tree they refer to
    - TLB does not have to be completely flushed at all if a unique identifier for each page table tree (i.e. a process's address space) is added in addition to the portion of the virtual address
    - kernel, VMM, and the individual processes all can have unique identifiers
    - only problem is that the number of bits available for the TLB tag is severely limited
        - number of address spaces is not
        - means some identifier reuse is necessary
        - TLB has to be partially flushed if possible
        - all entries with the reused identifier must be flushed but this is hopefully a much smaller set
    - useful outside the realm of virtualization when multiple processes are running on the system
        - good chance the most recently used TLB entries for a process are still in the TLB when it gets scheduled again if the memory use (and TLB entry use) of each of the runnable processes is limited
            a. special address spaces, e.g. those used by the kernel and VMM, are often only entered for a short time
                - control is often returned to the address space which initiated the entry
                - one or two TLB flushes are performed without tags
                - calling address space's cached translations are preserved with tags
                    - translations from previous system calls, etc. can still be used since the kernel and VMM address space do not often change TLB entries at all

            b. when switching between two threads of the same process no TLB flush is necessary at all
                - without extended TLB tags the entry into the kernel destroys the first thread's TLB entries
        - some processors have implemented these extended tags for a long time
        - AMD introduced a 1-bit tag extension with the Pacifica virtualization extensions
        - in the context of virtualization, 1-bit Address Space ID (ASID) is used to distinguish the VMM's address space from that of the guest domains
            - allows the OS to avoid flushing the guest's TLB entries every time the VMM is entered (e.g. to handle a page fault) or the VMM's TLB entries when control returns to the guest
            - architecture will allow the use of more bits in the future
            - other mainstream processors will likely follow suit and support this feature

4.3.2 Influencing TLB Performance

- first factor influencing TLB performance is the size of the pages
    - larger a page is, the more instructions or data objects will fit into it
    - larger page size reduces the overall number of address translations which are needed
        - means that fewer entries are needed in the TLB cache
    - most architectures nowadays allow the use of multiple different page sizes
        - some sizes can be used concurrently
            - x86/x86-64 processors have a normal page size of 4kB but they can also use 4MB and 2MB pages respectively
            - IA-64 and PowerPC allow sizes like 64kB as the base page size
    - problems with use of large page sizes
        - memory regions used for the large pages must be contiguous in physical memory
            - amount of wasted memory will grow if the unit size for the administration of physical memory is raised to the size of the virtual memory pages
            - all kinds of memory operations (like loading executables) require alignment to page boundaries
            - means, on average, that each mapping wastes half the page size in physical memory for each mapping
                - can easily add up
                    - puts an upper limit on the reasonable unit size for physical memory allocation
    - not practical to increase the unit size to 2MB to accommodate large pages on x86-64
        - means that each large page has to be comprised of many smaller pages
        - small pages have to be contiguous in physical memory
        - allocating 2MB of contiguous physical memory with a unit page size of 4kB can be difficult
            - requires finding a free area with 512 contiguous pages
            - after the system runs for a while and physical memory becomes fragmented this can be difficult or impossible
    - necessary to allocate these big pages at system start time using the special hugetlbfs filesystem on Linux
        - fixed number of physical pages are reserved for exclusive use as big virtual pages
            - ties down resources which might not always be used

      - limited pool
        - increasing it normally means restarting the system
  - huge pages are the way to go in situations when
    - performance is absolutely necessary
    - resources are plenty
    - difficult setup is ok
    - e.g. database servers
  - problems also occur when increasing the minimum virtual page size
    - memory mapping operations (e.g. loading applications) must conform to these page sizes
    - for most architectures location of the various parts of an executable have a fixed relationship
      - load operation cannot be performed if the page size is increased beyond what has been taken into account when the executable or DSO was built
    - alignment requirements of an ELF binary can be determined because they are encoded in the ELF program header
    - number of levels of the page table tree is reduced
      - not that many bits left which need to be handled through page directories ince the part of the virtual address corresponding to the page offset increases
        - amount of work which has to be done is reduced in case of a TLB miss
- second factor influencing TLB performance - possible to reduce the number of TLB entries needed by moving data which is used at the same time to fewer pages
  - similar to some optimizations for cache
  - requires large alignment
  - can be an important optimization given that the number of TLB entries is quite small this

```
eu-readelf -l /bin/ls
```

4.4 Impact Of Virtualization

- virtualization is more common and adds another layer of memory handling
  - virtualization of processes (basically jails) or OS containers do not fall into this category since only one OS is involved
- Xen, KVM, etc. enable the execution of independent OS images
  - one piece of software alone which directly controls access to the physical memory
- Xen VMM (hypervisor)
  - VMM does not implement many of the other hardware controls itself

    - hardware outside of memory and processors is controlled by the privileged Dom0 domain unlike VMMs on other, earlier systems (and the first release of the Xen VMM)
    - basically the same kernel as the unprivileged DomU kernels and do not differ as far as memory handling is concerned
    - NOTE: VMM hands out physical memory to the Dom0 and DomU kernels which then implement the usual memory handling as if they were running directly on a processor
- memory handling in the Dom0 and DomU kernels does not have unrestricted access to physical memory to implement the separation of the domains which is required for the virtualization to be complete
- VMM does not hand out memory by giving out individual physical pages and letting the guest OSes handle the addressing
    - would not provide any protection against faulty or rogue guest domains
- VMM creates its own page table tree for each guest domain and hands out memory using these data structures
- access to the administrative information of the page table tree can be controlled
    - if code does not have appropriate privileges it cannot do anything
- access control is exploited in the virtualization Xen provides
    - guest domains construct their page table trees for each process in a way which is intentionally quite similar for hardware virtualization
    - when the guest OS modifies its page tables the VMM is invoked
    - VMM then uses the updated information in the guest domain to update its own shadow page tables
    - these are the page tables which are actually used by the hardware
    - process is expensive
        - each modification of the page table tree requires an invocation of the VMM
- this cost is why the processors are starting to have additional functionality to avoid the creation of shadow page tables
    - good because of speed concerns
    - reduces memory consumption by the VMM
- Intel - Extended Page Tables (EPTs)
- AMD - Nested Page Tables (NPTs)
- both technologies have the page tables of the guest OSes produce "host virtual addresses" from the "guest virtual address"
    - host virtual addresses are then be further translated into actual physical addresses using the perdomain EPT/NPT trees
        - allows memory handling at almost the speed of the no-virtualization case since most VMM entries for memory handling are removed
        - reduces the memory use of the VMM since now only one page table tree for each domain (as opposed to process) has to be maintained
- results of the additional address translation steps are also stored in the TLB
    - TLB does not store the virtual physical address

    - instead stores the complete result of the lookup
- AMD's Pacifica extension introduced the ASID to avoid TLB flushes on each entry
    - number of bits for the ASID is one in the initial release of the processor extensions
        - just enough to differentiate VMM and guest OS
- Intel has virtual processor IDs (VPIDs) which serve the same purpose
    - more of them
    - VPID is fixed for each guest domain and therefore it cannot be used to mark separate processes and avoid TLB flushes at that level
- one problem with virtualized OSes is the amount of work needed for each address space modification
- no way around having two layers of memory handling (inherent in VMM-based virtualization)
- Xen approach of using a separate VMM makes optimal handling hard since all the complications of a memory management implementation must be duplicated in the VMM
    - OSes have fully-fledged and optimized implementations
        - avoid duplicating them
- for KVM Linux kernel extensions
    - no separate VMM running directly on the hardware and controlling all the guests
        - normal Linux kernel takes over this functionality
        - means the complete and sophisticated memory handling functionality in the Linux kernel is used to manage the memory of the system
    - guest domains run alongside the normal user-level processes in what the creators call "guest mode"
    - virtualization functionality, para- or full virtualization, is controlled by the KVM VMM
        - just another user-level process which happens to control a guest domain using the special KVM device the kernel implements
    - benefit over the separate VMM
        - only needs to be one implementation (one in Linux kernel) even though there are still two memory handlers at work when guest OSes are used
        - leads to less work, fewer bugs, and, perhaps, less friction where the two memory handlers touch since the memory handler in a Linux guest makes the same assumptions as the memory handler in the outer Linux kernel which runs on the bare hardware
- programmers must be aware that the cost of cache misses (instruction, data, or TLB) is even higher with virtualization than without virtualization
    - any optimization which reduces this work will pay off even more in virtualized environments
    - costs will be reduced by processor designers over time but will never go away

5 NUMA Support

- cost of access to specific regions of physical memory differs depending on where the access originated on some machines
    - this type of hardware requires special care from the OS and the applications

5.1 NUMA Hardware

- non-uniform memory architectures
- simplest form
    - processor can have local memory which is cheaper to access than memory local to other processors
- difference in cost for this type of NUMA system is not high, i.e., the NUMA factor is low
- used in big machines
- for commodity hardware all processors share the same Northbridge (excluding AMD Opteron NUMA)
    - makes the Northbridge a severe bottleneck since all memory traffic is routed through it
    - can use custom hardware in place of the Northbridge
        - unless the memory chips used have multiple ports (i.e. they can be used from multiple busse) there still is a bottleneck
    - multiport RAM is complicated and expensive to build and support
        - hardly ever used
- next step up in complexity is the model AMD uses where an interconnect mechanism (Hyper Transport) provides access for processors which are not directly connected to the RAM
    - size of the structures which can be formed this way is limited unless one wants to increase the diameter (i.e. maximum distance between any two nodes) arbitrarily
- efficient topology for connecting the nodes is the hypercube
    - limits the number of nodes to 2^C where C is the number of interconnect interfaces each node has
    - smallest diameter for all systems with 2^n CPUs and n interconnects
    - shows the first three hypercubes
    - each hypercube has a diameter of C which is the absolute minimum
    - AMD's first generation Opteron processors have three hypertransport links per processor
    - at least one of the processors has to have a Southbridge attached to one link
        - hypercube with C = 2 can be implemented directly and efficiently
        - next generation will at some point have four links, at which point C = 3 hypercubes will be possible

        - larger accumulations of processors can be supported
            - there are companies which have developed crossbars allowing larger sets of processors to be used (e.g. Newisys's Horus)
            - increases the NUMA factor and stop being effective at a certain number of processors
    - next step up - connecting groups of CPUs and implementing a shared memory for all of them
        - all such systems need specialized hardware and are by no means commodity systems
        - IBM x445 and similar machines are systems which are still quite close to commodity machines
        - interconnect used introduces a significant NUMA factor which the OS, as well as applications, must take into account At the other end of the spectrum,
        - machines like SGI's Altix are designed specifically to be interconnected
            - NUMAlink interconnect fabric is very fast and has low latency at the same time
                - requirements for high-performance computing (HPC)
                - specifically when Message Passing Interfaces (MPI) are used
            - drawback is that such sophistication and specialization is very expensive
            - make a reasonably low NUMA factor possible
            - with the number of CPUs these machines can have (several thousands) and the limited capacity of the interconnects, the NUMA factor is actually dynamic and can reach unacceptable levels depending on the workload
    - solutions where many commodity machines are connected using high-speed networking to form a cluster are more commonly
        - non-NUMA machines
            - do not implement a shared address space and therefore do not fall into any category which is discussed here

5.2 OS Support for NUMA

- OS has to take the distributed nature of the memory into account to support NUMA
    - physical RAM assigned to the process's address space should ideally come from local memory if a process is run on a given processor
        - otherwise each instruction has to access remote memory for code and data
- special cases to be taken into account which are only present in NUMA machines
- text segment of DSOs is normally present exactly once in a machine's physical RAM
    - if the DSO is used by processes and threads on all CPUs (e.g. basic runtime libraries like libc) means that all but a few processors have to have remote accesses
    - OS ideally would "mirror" such DSOs into each processor's physical RAM and use local copies
    - an optimization, not a requirement
    - hard to implement

    - might not be supported or only in a limited fashion To avoid making the situation worse, the
- OS should not migrate a process or thread from one node to another
    - OS should already try to avoid migrating processes on normal multi-processor machines because migrating from one processor to another means the cache content is lost
    - the OS can usually pick an arbitrary new processor which has sufficient capacity left if load distribution requires migrating a process or thread off of a processor
- in NUMA environments the selection of the new processor is more limited
    - newly selected processor should not have higher access costs to the memory the process is using than the old processor
        - restricts the list of targets
    - OS has no choice but to migrate to a processor where memory access is more expensive if there is no free processor matching that criteria available
    - two paths
        - if situation is temporary and the process can be migrated back to a better-suited processor
        - OS can also migrate the process's memory to physical pages which are closer to the newly used processor
            - expensive because possibly huge amounts of memory have to be copied
            - process at least briefly has to be stopped so that modifications to the old pages are correctly migrated
            - many other requirements for page migration to be efficient and fast
            - OS should avoid it unless it is really necessary
- cannot be assumed that all processes on a NUMA machine use the same amount of memory such that memory usage is also equally distributed with the distribution of processes across the processors
    - unless the applications running on the machines are very specific (common in the HPC world, but not outside) memory use will be unequal
    - leads to problems if memory is always allocated local to the processor where the request is originated
        - system will eventually run out of memory local to nodes running large processes
    - memory is, by default, not allocated exclusively on the local node in response to these problems
        - default strategy is to stripe the memory to utilize all the system's memory
            - guarantees equal use of all the memory of the system
        - becomes possible to freely migrate processes between processors since the access cost to all the memory used does not change on average
        - striping is acceptable but still not optimal for small NUMA factors
            - helps the system avoid severe problems and makes it more predictable under normal operation
            - decreases overall system performance

        - why Linux allows the memory allocation rules to be selected by each process
            - process can select a different strategy for itself and its children

5.3 Published Information

- kernel publishes information about the processor caches below through the sys pseudo file system (sysfs) - /sys/devices/system/cpu/cpu*/cache
- topology of the caches
    - directories in /sys/devices/system/cpu/cpu*/cache contain subdirectories (named index*) which list information about the various caches the CPU possesses
        - files type, level, and shared_cpu_map are the important files in these directories as far as the topology is concerned
        - see table for information layout on an Intel Core 2 QX6700
        - see table for information layout on an AMD Opteron
- sys file system exposes this information in the files in /sys/devices/system/cpu/cpu*/topology
- SMP Opteron machine is a NUMA machine
- another part of the sys file system which exists on NUMA machines - /sys/devices/system/node
    - contains information about the nature of NUMA on this machine
    - contains a subdirectory for every NUMA node on the system
    - in the node-specific directories there are a number of files
    - see table for important files and their content for the Opteron
- using these files can construct complete picture of the architecture of the machine
- each processor constitutes its own node as can be seen by the bits set in the value in cpumap file in the node* directories
    - distance files in those directories contains a set of values, one for each node, which represent a cost of memory accesses at the respective nodes

5.4 Remote Access Costs

- distance is relevant
- writes are slower than reads
- 2-hop reads and writes are 30% and 49% (respectively) slower than 0-hop reads
- 2- hop writes are 32% slower than 0-hop writes, and 17% slower than 1-hop writes
- relative position of processor and memory nodes can make a big difference
- next generation of processors from AMD will feature four coherent HyperTransport links per processor
- in that case a four socket machine would have diameter of one
- with eight sockets the same problem returns, with a vengeance, since the diameter of a hypercube with eight nodes is three

- possible to determine how the memory-mapped files, the Copy-On-Write (COW) pages and anonymous memory are distributed over the nodes in the system
    - Copy-On-Write is a method often used in OS implementations when a memory page has one user at first and then has to be copied to allow independent users
        - in many situations the copying is unnecessary, at all or at first, in which case it makes sense to only copy when either user modifies the memory
        - operating system intercepts the write operation, duplicates the memory page, and then allows the write instruction to proceed
- for each process the kernel provides a pseudo-file /proc/PID/numa_maps
    - important information in the file is the values for N0 to N3, which indicate the number of pages allocated for the memory area on nodes 0 to 3
    - program itself and the dirtied pages are allocated on that node
    - read-only mappings, such as the first mapping for ld-2.4.so and libc-2.4.so as well as the shared file locale-archive are allocated on other nodes
- when performed across nodes the read performance falls by 9% and 30% respectively for 1- and 2-hop reads
    - for execution, such reads are needed and, if the L2 cache is missed, each cache line incurs these additional costs
    - all the costs measured for large workloads beyond the size of the cache would have to be increased by 9%/30% if the memory is remote to the processor
- can measure the bandwidth with the memory being on a remote node, one hop away, to see effects in real world
    - result of this test when compared with the data for using local memory
    - numbers have a few big spikes in both directions which are the result of a problem of measuring multi-threaded code and can be ignored
    - important information is that read operations are always 20% slower
        - significantly slower than the 9% which is, most likely, not a number for uninterrupted read/write operations and might refer to older processor revisions
    - for working set sizes which fit into the caches, the performance of write and copy operations is also 20% slower
    - for working sets exceeding the size of the caches, the write performance is not measurably slower than the operation on the local node
    - speed of the interconnect is fast enough to keep up with the memory
    - dominating factor is the time spent waiting on the main memory

6 What Programmers Can Do

- many opportunities for programmers to influence a program's performance, positively or negatively
- starting with the lowest levels of physical RAM access and L1 caches, up to and including OS functionality which influences memory handling

6.1 Bypassing the Cache

- memory store operations read a full cache line first and then modify the cached data
    - when data is produced and not (immediately) consumed again this fact is detrimental to performance
    - operation pushes data out of the caches which might be needed again in favor of data which will not be used soon
    - especially true for large data structures, like matrices, which are filled and then used later
        - before the last element of the matrix is filled the sheer size evicts the first elements, making caching of the writes ineffective
- processors provide support for non-temporal write operations for above and similar situations
    - means the data will not be reused soon, so there is no reason to cache it
    - non-temporal write operations do not read a cache line and then modify it
        - new content is directly written to memory
    - does not have to be expensive
    - processor will try to use write-combining to fill entire cache lines
        - if this succeeds no memory read operation is needed at all
- a number of intrinsics are provided by gcc for similar situations

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);

#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);

#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

- used most efficiently if they process large amounts of data in one go
    - data is loaded from memory
    - processed in one or more steps
    - then written back to memory
    - data "streams" through the processor
- memory address must be aligned to 8 or 16 bytes respectively
    - in code using the multimedia extensions it is possible to replace the normal *mm_store* * intrinsics with non-temporal versions

- processor's write-combining buffer can hold requests for partial writing to a cache line for only so long
    - generally necessary to issue all the instructions which modify a single cache line one after another so that the write-combining can actually take place

```
#include <emmintrin.h>
void setbytes(char *p, int c)
{
    __m128i i = _mm_set_epi8(c, c, c, c, c, c, c, c, c, c, c, c, c, c, c, c);
    _mm_stream_si128((__m128i *)&p[0], i);
    _mm_stream_si128((__m128i *)&p[16], i);
    _mm_stream_si128((__m128i *)&p[32], i);
    _mm_stream_si128((__m128i *)&p[48], i);
}
```

- call to this function will set all bytes of the addressed cache line to c (if pointer p is appropriately aligned)
    - write-combining logic will see the four generated movntdq instructions and only issue the write command for the memory once the last instruction has been executed
    - code sequence not avoids reading the cache line before it is written
    - also avoids polluting the cache with data which might not be needed soon
    - memset function in the C runtime is an example of everyday code using this technique
        - should use a code sequence like the above for large blocks Some architectures provide specialized solutions. The
- PowerPC architecture defines the dcbz instruction which can be used to clear an entire cache line
- see text for another test for non-temporal instructions
    - mainly interested in are writes bypassing the cache
        - the sequential access is just as fast here as in the case where the cache is used
        - the processor is performing write-combining
        - memory ordering rules for non-temporal writes are relaxed
            - program needs to explicitly insert memory barriers (sfence instructions for the x86 and x86-64 processors)
            - means the processor has more freedom to write back the data and thereby using the available bandwidth as well as possible
- case of column-wise access
    - results for uncached accesses are significantly slower than in the case of cached accesses
    - no write combining is possible and each memory cell must be addressed individually
        - requires constantly selecting new rows in the RAM chips with all the associated delays
        - result is a 25% worse result than the cached run On the read side,

- processors, until recently, lacked support aside from weak hints using non-temporal access (NTA) prefetch instructions for the read side
  - no equivalent to write-combining for reads
    - bad for uncacheable memory such as memory-mapped I/O
  - Intel introduced NTA loads with the SSE4.1 extensions
    - implemented using a small number of streaming load buffers
      - each buffer contains a cache line
      - first movntdqa instruction for a given cache line will load a cache line into a buffer, possibly replacing another cache line
      - subsequent 16-byte aligned accesses to the same cache line will be serviced from the load buffer at little cost
      - unless there are other reasons to do so, the cache line will not be loaded into a cache
      - enables the loading of large amounts of memory without polluting the caches
      - compiler provides an intrinsic for this instruction

```
#include <smmintrin.h>
__m128i _mm_stream_load_si128 (__m128i *p);
```

- intrinsic should be used multiple times
  - addresses of 16-byte blocks passed as the parameter
  - used until each cache line is read
  - next cache line should be started
  - there are a few streaming read buffers it might be possible to read from two memory locations at once
- modern CPUs optimize uncached write well
- modern CPUs optimize read accesses as long as they are sequential
  - useful when handling large data structures which are used only once
- caches can help to cover up some–but not all–of the costs of random memory access
  - random access in this example is 70% slower due to the implementation of RAM access
  - random accesses should be avoided whenever possible until the implementation changes

6.2 Cache Access

- best to focus on changes affecting the level 1 cache
- all the optimizations for the level 1 cache also affect the other caches
- same approach for all memory
  - improve locality (spatial and temporal) and align the code and data

6.2.1 Optimizing Level 1 Data Cache Access

- example code is matrix multiplication
  - two square matrices of 1000 x 1000 double

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        for (k = 0; k < N; ++k)
            res[i][j] += mul1[i][k] * mul2[k][j];
```

- the inner loop advances the row number of mul2 while mul1 is accessed sequentially
  - causes problems
- worthwhile to rearrange the second matrix mul2 before using it because each element in the matrices is accessed multiple times
- iterate over both matrices sequentially after the transposition (traditionally indicated by a superscript T)

```
double tmp[N][N];
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        tmp[i][j] = mul2[j][i];
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        for (k = 0; k < N; ++k)
            res[i][j] += mul1[i][k] * tmp[j][k];
```

- create a temporary variable to contain the transposed matrix
  - requires touching additional memory but probably recovered
  - 1000 nonsequential accesses per column are more expensive
- results on a Intel Core 2 with 2666MHz clock speed show a 76.6% speed up
  - 1000 nonsequential accesses cause a serious impact
- matrix can be too large or the available memory too small
  - cannot always perform the copy
- start with a close examination of the math involved and the operations performed by the original implementation
  - order in which the additions for each element of the result matrix are performed is irrelevant as long as each addend appears exactly once
  - look for solutions which reorder the additions performed in the inner loop of the original code Now let us examine the actual problem in the execution of the original code. The
- order in which the elements of mul2 are accessed is: (0,0), (1,0), . . . , (N -1,0), (0,1), (1,1), ... The
- each round of the inner loop requires, for each of the three matrices, 1000 cache lines (64 bytes for the Core 2 processor)
  - adds up to much more than the 32k of L1d available

- handling two iterations of the middle loop together while executing the inner loop
    - leads to two double values from the cache line which is guaranteed to be in L1d
    - cuts the L1d miss rate in half which is an improvement
    - can improve more depending on the cache line size
    - Core 2 processor has a L1d cache line size of 64 bytes
    - actual value can be queried using `sysconf(_SC_LEVEL1_DCACHE_LINESIZE)` at runtime or using the getconf utility from the command line so that the program can be compiled for a specific cache line size
    - unroll the middle loop 8 times to fully utilize the cache line because sizeof(double) is 8
    - to write 8 results at the same time to effectively use the res matrix should unroll the outer loop 8 times as well
        - assume cache lines of size 64
        - code works also well on systems with 32 byte cache lines since both cache lines are also 100% utilize
- best to hardcode cache line sizes at compile time by using the getconf utility `gcc -DCLS=$(getconf LEVEL1_DCACHE_LINESIZE) ...`
- largest cache line size should be used if binaries are supposed to be generic
    - might mean that not all the data fits into the cache with very small L1ds
        - such processors are not suitable for high-performance programs

```
#define SM (CLS / sizeof (double))
for (i = 0; i < N; i += SM)
    for (j = 0; j < N; j += SM)
        for (k = 0; k < N; k += SM)
            for (i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k]; i2 < SM; ++i2, rr
                for (k2 = 0, rmul2 = &mul2[k][j]; k2 < SM; ++k2, rmul2 += N)
                    for (j2 = 0; j2 < SM; ++j2)
                        rres[j2] += rmul1[k2] * rmul2[j2];
```

- most visible change is now have six nested loops

    - outer loops iterate with intervals of SM (the cache line size divided by sizeof(double))
    - divides the multiplication in several smaller problems which can be handled with more cache locality
    - inner loops iterate over the missing indexes of the outer loops
    - k2 and j2 loops are in a different order
        - only one expression depends on k2 but two depend on j2 in the actual computation
    - rest of the complication here results from the fact that gcc is not very smart when it comes to optimizing array indexing
    - introduction of the additional variables rres, rmul1, and rmul2 optimizes the code by pulling common expressions out of the inner loops, as far down as possible

    - default aliasing rules of the C and C++ languages do not help the compiler making these decisions (unless restrict is used, all pointer accesses are potential sources of aliasing)
        - why Fortran is still a preferred language for numeric programming
            - makes writing fast code easier
    - restrict keyword introduced into the C language in the 1999 revision should solve the problem
        - compilers have not caught up yet
        - reason is mainly that too much incorrect code exists which would mislead the compiler and cause it to generate incorrect object code

- gain another 6.1% of performance by avoiding the copying
    - do not need any additional memory
    - input matrices can be arbitrarily large as long as the result matrix fits into memory as well
    - requirement for a general solution

- in all cases there are no negative effects if the working set fits into L1d
    - once L1d is no longer sufficient, penalties are paid by using two cache lines in the process instead of one
    - about 17% penalty when the L2 cache is sufficient and about 27% penalty when the main memory has to be used
        - shows the data when the list is laid out sequentially in memory

- relative data for memory accesses
    - slowdown for working sets which fit into L2 is between 25% and 35%
    - is not because the penalties get smaller but, instead,
    - memory accesses get disproportionally more costly
    - in some cases the distance between the elements does matter

- easy way to see the layout of a data structure compared to cache lines is to use the pahole program
    - program examines the data structures defined in a binary

- pahole output on 64-bit machine shows
    - data structure uses up more than one cache line
    - tool assumes the currently used processor's cache line the structure size
    - value can be overridden using a command line parameter

  - makes sense to seek a way to compress a structure in cases where the size of the structure is barely over the limit of a cache line and many objects of this type are allocated
    - a few elements can have a smaller type
    - some fields are actually flags which can be represented using individual bits
    - etc.

- in example program output shows that there is a hole of four bytes after the first element
  - caused by the alignment requirement of the structure and the fill element
  - element b, which has a size of four bytes (indicated by the 4 at the end of the line), fits perfectly into the gap
  - when gap no longer exists and that the data structure fits onto one cache line
  - use --reorganize parameter is used and the structure name is added at the end of the command line the output of the tool is the optimized structure and the cache line use
  - tool can also optimize bit fields and combine padding and holes

- position of the individual structure elements and the way they are used is important
  - performance of code with the critical word late in the cache line is worse

- programmer should always follow these rules
  i. move the structure element which is most likely to be the critical word to the beginning of the structure
  ii. access the elements in the order in which they are defined in the structure when accessing the data structures if order of access is not dictated by the situation

- elements should be arranged in the order in which they are likely accessed for small structures

- each cache line-sized block should be arranged to follow the rules or bigger data structures

- reordering elements is not worth the time it takes if the object itself is not aligned as expected
  - alignment of an object is determined by the alignment requirement of the data type
  - each fundamental type has its own alignment requirement
  - largest alignment requirement of any of its elements determines the alignment of the structure for structured types
    - almost always smaller than the cache line size
  - even if the members of a structure are lined up to fit into the same cache line an allocated object might not have an alignment matching the cache line size

    - two ways to ensure that the object has the alignment which was used when designing the layout of the structure:
        - object can be allocated with an explicit alignment requirement
            - for dynamic allocation a call to malloc would only allocate the object with an alignment matching that of the most demanding standard type (usually long double)
            - possible to use posix_memalign to request higher alignments
        - alignment requirement of a user-defined type can be changed by using a type attribute
            - will cause the compiler to allocate all objects with the appropriate alignment, including arrays
            - programmer has to take care of requesting the appropriate alignment for dynamically allocated objects
            - posix_memalign must be used
            - use the alignof operator gcc provides and pass the value as the second parameter to posix_memalign

- multimedia extensions previously mentioned in this section almost always require that the memory accesses are aligned

    - for 16 byte memory accesses the address is supposed to be 16 byte aligned
    - x86 and x86-64 processors have special variants of the memory operations which can handle unaligned accesses but these are slower
    - hard alignment requirement is nothing new for most RISC architectures
        - require full alignment for all memory accesses
    - if an architecture supports unaligned accesses this is sometimes slower than using appropriate alignment
        - especially if the misalignment causes a load or store to use two cache lines instead of one

- effects of unaligned memory accesses

    - slowdown a program incurs because of the unaligned accesses
        - effects are more dramatic for the sequential access case than for the random case
            - costs of unaligned accesses are partially hidden by the generally higher costs of the memory access for random accesses
            - for working set sizes which do fit into the L2 cache slowdown is about 300% for sequential case

- effects of the unaligned access are still 20% to 30% for very large working set sizes

- negative results of alignment requirements

  - the compiler has to ensure that it is met in all situations if an automatic variable has an alignment requirement
    - not trivial since the compiler has no control over the call sites and the way they handle the stack
      1. generated code actively aligns the stack, inserting gaps if necessary
         - requires code to check for alignment, create alignment, and later undo the alignment
      2. require that all callers have the stack aligned

- all of the commonly used application binary interfaces (ABIs) number 2

- size of a stack frame used in a function is not necessarily a multiple of the alignment

  - padding is needed if other functions are called from this stack frame
  - the stack frame size is, in most cases, known to the compiler
    - knows how to adjust the stack pointer to ensure alignment for any function which is called from that stack frame
  - most compilers will simply round the stack frame size up
  - not possible if variable length arrays (VLAs) or alloca are used
    - total size of the stack frame is only known at runtime
    - active alignment control might be needed in this case, making the generated code (slightly) slower

- only the multimedia extensions require strict alignment on some architectures

  - stacks are always minimally aligned for the normal data types
    - usually 4 or 8 byte alignment for 32and 64-bit architectures respectively
    - enforcing the alignment incurs unnecessary costs
    - might want to get rid of the strict alignment requirement if is known that it is never depended upon
  - tail functions (those which call no other functions) which do no multimedia operations do not need alignment
  - functions which only call functions which need no alignment do not need alignment
  - program might want to relax the alignment requirement if a large enough set of functions can be identified
  - gcc has support for relaxed stack alignment requirements for x86 binaries: `-mpreferred-stack-boundary=2`
    - given a value of N, the stack alignment requirement will be set to 2N bytes
    - no additional alignment operation in most cases
      - needed since normal stack push and pop operations work on four-byte boundaries anyway

    - machine-specific option can help to reduce code size and also improve execution speed
        - cannot be applied for many other architectures
        - generally not applicable since the x86-64 ABI requires that floating-point parameters are passed in an SSE register and the SSE instructions require full 16 byte alignment even for x86-64

- if an array of structures is used the entire structure definition affects performance
    - might be necessary to rearrange data structures
    - can split data structures into separate pieces
        - increases the complexity of the program but the performance gains might justify this cost

- another cache use optimization which is primarily felt in the L1d access (also applies to the other caches)
    - an increased associativity of the cache benefits normal operation
    - the larger the cache, the higher the associativity usually is
    - L1d cache is too large to be fully associative but not large enough to have the same associativity as L2 caches
    - can be a problem if many of the objects in the working set fall into the same cache set
    - if this leads to evictions due to overuse of a set, the program can experience delays even though much of the cache is unused
        - these cache misses are sometimes called conflict misses
    - programmer can control since the L1d addressing uses virtual addresses
    - if variables which are used together are also stored together the likelihood of them falling into the same set is minimized

- set associativity is something worth paying attention to
    - laying out data at boundaries that are powers of two happens often in the real world
        - exactly the situation which can easily lead to described effects and degraded performance
    - unaligned accesses can increase the probability of conflict misses since each access might require an additional cache line

- another possible optimization
    - AMD's processors implement the L1d as several individual banks
    - L1d can receive two data words per cycle but only if both words are stored in different banks or in a bank with the same index
    - bank address is encoded in the low bits of the virtual
    - if variables which are used together are also stored together the likelihood that they are in different banks or the same bank with the same index is high

6.2.2 Optimizing Level 1 Instruction Cache Access

6.2.3 Optimizing Level 2 and Higher Cache Access

6.3 Prefetching

6.3.1 Hardware Prefetching

6.3.2 Software Prefetching

6.3.4 Helper Threads

6.3.5 Direct Cache Access

6.4 Multi-Thread Optimizations

6.4.1 Concurrency Optimizations

6.4.2 Atomicity Optimization

6.4.3 Bandwidth Considerations

6.5 NUMA Programming

6.5.1 Memory Policy

6.5.2 Specifying Policies

6.5.3 Swapping and Policies

6.5.4 VMA Policy

6.5.5 Querying Node Information

6.5.6 CPU and Node Sets

6.5.8 Utilizing All Bandwidth

7 Memory Performance Tools

7.1 Memory Operation Profiling

7.2 Simulating CPU Caches

7.4 Improving Branch Prediction

7.5 Page Fault Optimization

8 Upcoming Technology

8.1 The Problem with Atomic Operations

8.2 Transactional Memory

8.2.1 Load Lock/Store Conditional Implementation

8.2.2 Transactional Memory Operations

8.2.3 Example Code Using Transactional Memory

8.2.4 Bus Protocol for Transactional Memory

8.2.5 Other Considerations

8.3 Increasing Latency

8.4 Vector Operations

A Examples and Benchmark Programs A.1 Matrix Multiplication

B Some OProfile Tips

B.1 Oprofile Basics

B.2 How It Looks Like

B.3 Starting To Profile

C Memory Types

Though it is not necessary knowledge for efficient programming, it might be useful to describe some more technical details of available memory types. Specifically we are here interested in the

- difference of "registered" versus "unregistered" and ECC versus non-ECC DRAM types
- "registered" and "buffered" are used synonymously when describing a DRAM type which adds a buffer to the DRAM module
- all DDR memory types can come in registered and unregistered form
- for the unregistered modules, the memory controller is directly connected to all the chips on the module
- demanding electrically
  - memory controller must be able to deal with the capacities of all the memory chips
  - not ideal if the memory controller (MC) has a limitation, or if many memory modules are to be used
- buffered (or registered) memory
  - connected to a buffer instead of directly connecting the RAM chips on the DRAM module to the memory
  - buffer is then connected to the memory controller
  - significantly reduces the complexity of the electrical connections
- ability of the memory controllers to drive DRAM modules increases by a factor corresponding to the number of connections saved
- why aren't all DRAM modules MC buffered?
  - buffered modules are more complicated and more expensive
  - buffer delays the signals from the RAM chips a bit
    - delay must be high enough to ensure that all signals from the RAM chips are buffered
  - result is that the latency of the DRAM module increases
  - additional electrical component increases the energy cost
    - since the buffer has to operate at the frequency of the bus this component's energy consumption can be significant

- with the other factors of the use of DDR2 and DDR3 modules it is usually not possible to have more than two DRAM modules per bank
  - number of pins of the memory controller limit the number of banks (to two in commodity hardware)
- most memory controllers are able to drive four DRAM modules and, therefore, unregistered modules are sufficient
- different in server environments
  - errors are possible due to the minuscule charges held by the capacitors in the RAM cells
  - natural phenomena lead to errors where the content of RAM cell changes from 0 to 1 or vice versa
  - higher the probability of such an event when more memory is used
- ECC (Error Correction Code) DRAM can be used if such errors are not acceptable
  - enable the hardware to recognize incorrect cell contents and, in some cases, correct the errors
  - parity checks used to be used to recognized errors
    - machine had to be stopped when one was detected
  - small number of erroneous bits can be automatically corrected with ECC
  - if the number of errors is too high, though, the memory access cannot be performed correctly and the machine still stops
    - unlikely case for working DRAM modules, though, since multiple errors must happen on the same module
- memory controller and not the memory which performs the error checking
  - DRAM modules simply provide more storage and transport the additional non-data bits along with the real data
  - ECC memory stores one additional bit for each 8 data bits
  - on writing data to a memory address, the memory controller computes the ECC for the new content on the fly before sending that data and ECC onto the memory bus
  - when reading, the data plus the ECC is received, the memory controller computes the ECC for the data, and compares it with the ECC transmitted from the DRAM module
  - if the ECCs match everything is fine. If they do not match, the memory controller tries to correct the error
  - if this correction is not possible, the error is logged and the machine is possibly halted Several techniques for error correction are in use but, for DRAM ECC, usually
- Hamming codes are usually used for DRAM ECC
  - originally were used to encode four data bits with the ability to recognize and correct one flipped bit (SEC, Single Error Correction)
  - can easily be extended to more data bits
  - relationship between the number of data bits W and the number of bits for the error code E is described by the equation

```
E = [log2(W + E + 1)]
```

- pn most modules, each RAM chip produces 8 bits and, therefore, any other combination would lead to less efficient solution
- with more effort is it possible to create codes which can correct two flipped bits and more
    - probability and risk which decide whether this is needed
- some memory manufacturers say an error can occur in 256MB of RAM every 750 hours
    - by doubling the amount of memory the time is reduced by 75%
- with enough memory the probability of experiencing an error in a short time can be significant and ECC RAM becomes a requirement
- time frame could even be so small that the SEC/DED implementation is not sufficient.
- server motherboards have the ability to automatically read all memory over a given timeframe instead of implementing even more error correction capabilities
    - memory controller reads the data and, if the ECC check fails, writes the corrected data back to memory whether or not the memory was actually requested by the processor
    - as long as the probablity of incurring less than two memory errors in the time frame needed to read all of memory and write it back is acceptable, SEC/DED error correction is a perfectly reasonable solution
- why is ECC DRAM not the norm?
    - extra RAM chip increases the cost and the parity computation increases the delay
    - unregistered, non-ECC memory can be significantly faster
    - because of the similarity of the problems of registered and ECC DRAM, one usually only finds registered, ECC DRAM and not registered, non-ECC DRAM
- some manufacturers offer what is often incorrectly called "memory RAID" where the data is distributed redundantly over multiple DRAM modules, or at least RAM chips
    - motherboards with this feature can use unregistered DRAM modules, but the increased traffic on the memory busses is likely to negate the difference in access times for ECC and non-ECC DRAM modules

D libNUMA Introduction

http://people.redhat.com/drepper/libNUMA.tar.bz2