Computer Hardware

MJ Rutter

mjr19@cam.ac.uk

Michaelmas 2013

©2013 MJ Rutter

Computer Hardware

MJ Rutter
mjr19@cam

Michaelmas 2013

Bibliography

Computer Architecture, A Qualitative Approach, 5th Ed., Hennessy, JL and Patterson, DA, pub. Morgan Kaufmann, c.£40.
Usually considered the standard textbook on computer architecture, and kept reasonably up-to-date. The fifth edition was published in 2011, although much material in earlier editions is still relevant, and early editions have more on paper, and less on CD / online, though with 850 pages, there is quite a lot on paper...

Typeset by FoilTEX ©2013 MJ Rutter

Contents

History 4

The CPU 10

- instructions . . . 17
- pipelines . . . 18
- vector computers . . . 36
- performance measures . . . 39

Memory 44

- DRAM . . . 45
- caches . . . 57

Memory Access Patterns in Practice 84

- matrix multiplication . . . 84
- matrix transposition . . . 109

Memory Management 120

- virtual addressing . . . 121
- paging to disk . . . 130
- memory segments . . . 139

Hello: My First Program 160

- direct hardware access . . . 161
- using the OS . . . 169
- using libraries . . . 176

Compilers & Optimisation 196

- optimisation . . . 197
- the pitfalls of F90 . . . 221

Disks, Filesystem & Fileservers 234

- disks . . . 235
- RAID . . . 244
- filesystems . . . 247
- fileservers . . . 265

Parallel Computers 274

- multitasking . . . 274
- parallel computers . . . 278
- MPP . . . 282
- SMP . . . 289
- NUMA . . . 294

Index 315

History

History: to 1970

1951	Ferranti Mk I: first commercial computer
	UNIVAC I: memory with parity
1953	EDSAC I ‘heavily used’ for science (Cambridge)
1954	Fortran I (IBM)
1955	Floating point in hardware (IBM 704)
1956	Hard disk drive prototype. $24''$ platters (IBM)
1961	Fortran IV
	Pipelined CPU (IBM 7030)
1962	Hard disk drive with flying heads (IBM)
1963	CTSS: Timesharing (multitasking) OS
	Virtual memory & paging (Ferranti Atlas)
1964	First BASIC
1967	ASCII character encoding (current version)
	GE635 / Multics: SMP (General Elect)
1968	Cache in commercial computer (IBM 360/85)
	Mouse demonstrated
	Reduce: computer algebra package
1969	ARPAnet: wide area network
	Fully pipelined functional units (CDC 7600)
	Out of order execution (IBM 360/91)

1970 First DRAM chip. 1Kbit. (Intel)
First floppy disk. 8″ (IBM)
1971 UNIX appears within AT&T
Pascal
First email
1972 Fortran 66 standard published
First vector computer (CDC)
First TLB (IBM 370)
ASC: computer with 'ECC' memory (TI)
1973 First 'Winchester' (hard) disk (IBM)
1974 First DRAM with one transistor per bit
1975 UNIX appears outside AT&T
Ethernet appears (Xerox)
1976 Apple I launched. $666.66
Cray I, ILLIAC IV
Z80 CPU (used in Sinclair ZX series) (Zilog)
$5\frac{1}{4}''$ floppy disk
1978 K&R C appears (AT&T)
TCP/IP
Intel 8086 processor
Laser printer (Xerox)
WordStar (early wordprocessor)
First VAX (11/780) and VMS (DEC)
1979 TeX

1980 Sinclair ZX80 £100 (10^5 sold eventually)
Fortran 77 standard published
1981 Sinclair ZX81 £70 (10^6 sold eventually)
$3\frac{1}{2}''$ floppy disk (Sony)
IBM PC & MS DOS version 1 $3,285
SMTP (current email standard) proposed
1982 Sinclair ZX Spectrum £175 48KB colour
Acorn BBC model B £400 32KB colour
Commodore64 $600 ($10^7$ sold eventually)
Cray X-MP (first multiprocessor Cray)
Motorola 68000 (commodity 32 bit CPU)
1983 Internet defined to be TCP/IP only
Apple IIe $1,400
IBM XT, $7,545
Caltech Cosmic Cube: 64 node 8086/7 MPP
1984 Apple Macintosh $2,500. 128KB, 9" B&W screen
Sinclair QL £400. 128KB
IBM AT, $6,150. 256KB
CD ROM
1985 LaTeX2.09
PostScript (Adobe)
Ethernet formally standardised
IEEE 748 formally standardised
Intel i386 (Intel's first 32 bit CPU)
X10R1 (forerunner of X11) (MIT)
C++

History: the RISCs

1986	MIPS R2000, RISC CPU (used by SGI and DEC)
	SCSI formally standardised
1987	Intel i860 (Intel's first RISC CPU)
	Acorn Archimedes (ARM RISC) £800
	SPARC I, RISC CPU (Sun)
	Macintosh II $4,000. FPU and colour.
	Multiflow Trace/200: VLIW CPU
	X11R1 (MIT)
1989	ANSI C
1990	PostScript Level 2
	Power I: superscalar RISC (IBM)
	MS Windows 3.0
1991	World Wide Web / HTTP
	PVM (later superceded by MPI)
	Fortran 90
1992	PCI
	OpenGL
	OS/2 2.0 (32 bit a year before Windows NT) (IBM)
	Alpha 21064: 64 bit superscalar RISC CPU (DEC)

A Summary of History

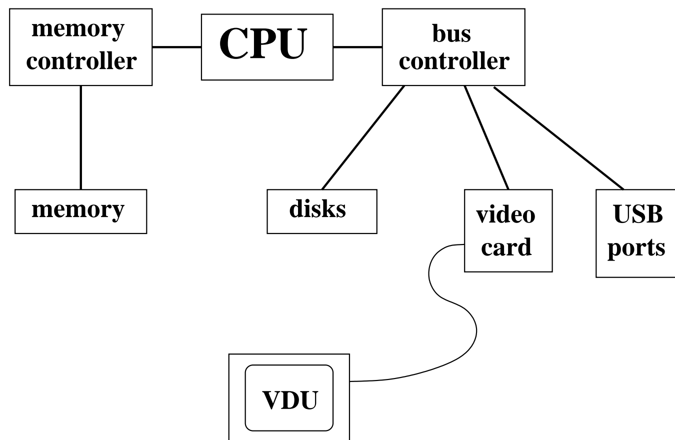
The above timeline stops about two decades before a precursor to this talk was first given. Computing is not a fast-moving subject, and little of consequence has happened since. . .

By 1970 the concepts of disk drives, floating point, memory paging, parity protection, multitasking, caches, pipelining and out of order execution have all appeared in commercial systems, and high-level languages and wide area networking have been developed. The 1970s themselves add vector computers and error correcting memory, and implicit with the vector computers, RISC.

The rest is just enhanced technology rather than new concepts. The 1980s see the first serious parallel computers, and much marketing in a home computer boom. The slight novelty to arrive in the 21st century is the ability of graphics cards to do floating point arithmetic, and to run (increasingly complex) programs. ATI's 9700 (R300) launched in late 2002 supported FP arithmetic. Nvidia followed a few months later.

The CPU

Inside the Computer



The Heart of the Computer

The CPU, which for the moment we assume has a single core, is the brains of the computer. Everything else is subordinate to this source of intellect.

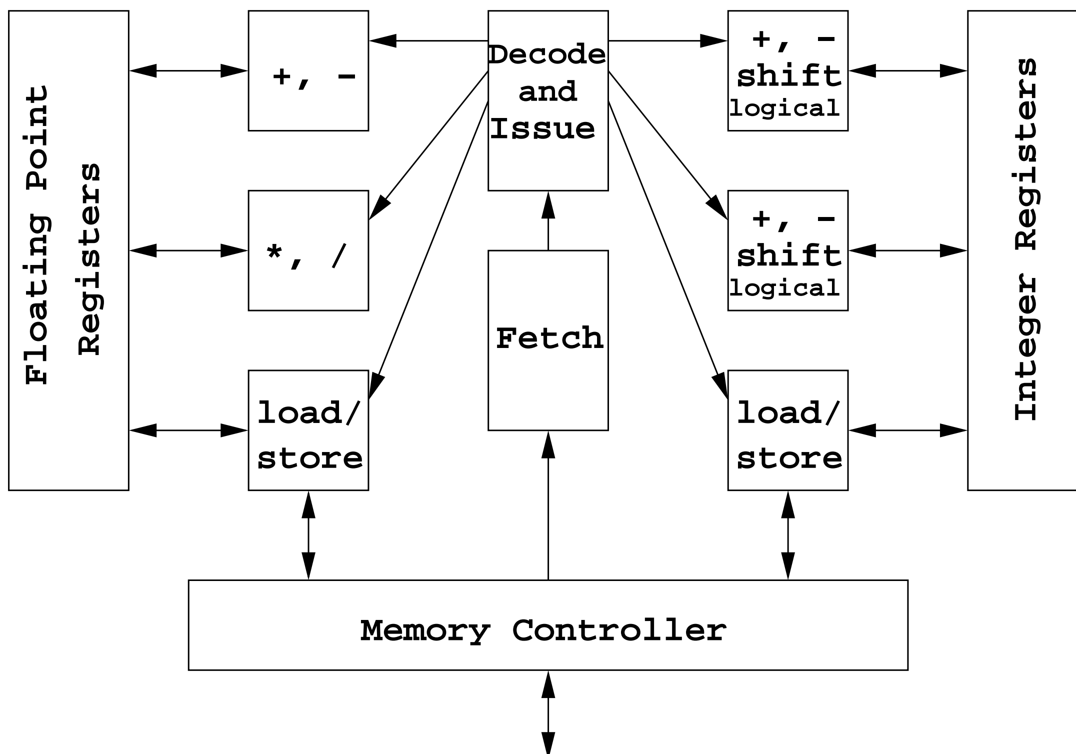
A typical modern CPU understands two main classes of data: integer and floating point. Within those classes it may understand some additional subclasses, such as different precisions.

It can perform basic arithmetic operations and comparisons, governed by a sequence of instructions, or *program*.

It can also perform comparisons, the result of which can change the *execution path* through the program.

Its sole language is machine code, and each family of processors speaks a completely different variant of machine code.

Schematic of Typical RISC CPU



What the bits do

- Memory: not part of the CPU. Used to store both program and data.
- Instruction fetcher: fetches next machine code instruction from memory.
- Instruction decoder: decodes instruction, and sends relevant data on to…
- Functional unit: dedicated to performing a single operation
- Registers: store the input and output of the functional units There are typically about 32 floating point registers, and 32 integer registers.

Partly for historical reasons, there is a separation between the integer and floating point parts of the CPU.

On some CPUs the separation is so strong that the only way of transferring data between the integer and floating point registers is via the memory. On some older CPUs (e.g. the Intel 386), the FPU (floating point unit) is optional and physically distinct.

Clock Watching

The best known part of a CPU is probably the *clock*. The clock is simply an external signal used for synchronisation. It is a square wave running at a particular frequency.

Clocks are used within the CPU to keep the various parts synchronised, and also on the data paths between different components external to the CPU. Such data paths are called *buses*, and are characterised by a *width* (the number of wires (i.e. bits) in parallel) as well as a clock speed. External buses are usually narrower and slower than ones internal to the CPU.

Although sychronisation is important – every good orchestra needs a good conductor – it is a means not an end. A CPU may be designed to do a lot of work in one clock cycle, or very little, and comparing clock rates between different CPU designs is meaningless.

The bandwidth of a bus is simple its width $\times$ its clock speed $\times$ the number of data transfers per clock cycle. For the original IBM PC bus, 1 byte $\times$ 4.77MHz $\times$ one quarter (1.2MB/s). For PCIe v2 x16, 2 bytes $\times$ 5GHz $\times$ four fifths (8GB/s).

Work Done per Clock Cycle

Intel's 'Sandy Bridge' and 'Ivy Bridge' range of CPUs can theoretically sustain four floating point adds and four floating point multiplies per core per clock-cycle.

The previous generations (Nehalem and Core2) just two adds and two multiplies.

The previous generation (Pentium4) just one add and one multiply.

The previous generations (Pentium to Pentium III) just one add or multiply.

The previous generation (i486), about a dozen clock cycles for one FP add or multiply. The generation before (i386/i387) about two dozen cycles.

Since the late 1980s clock speeds have improved by a factor of about 100 (c. 30MHz to c. 3GHz). The amount of floating point work a single core can do in one clock cycle has also increased by a factor of about 100.

Typical instructions

Integer:

- arithmetic: $+, -, \times, /$, negate
- logical: and, or, not, xor
- bitwise: shift, rotate
- comparison
- load / store (copy between register and memory)

Floating point:

- arithmetic: $+, -, \times, /, \surd$, negate, modulus
- convert to / from integer
- comparison
- load / store (copy between register and memory)

Control:

- (conditional) branch (i.e. goto)

Most modern processors barely distinguish between integers used to represent numbers, and integers used to track memory addresses (i.e. pointers).

A typical instruction

```
fadd f4,f5,f6
```

add the contents of floating point registers 4 and 5, placing the result in register 6.

Execution sequence:

- fetch instruction from memory
- decode it
- collect required data (`f4` and `f5`) and send to floating point addition unit
- wait for add to complete
- retrieve result and place in `f6`

Exact details vary from processor to processor.

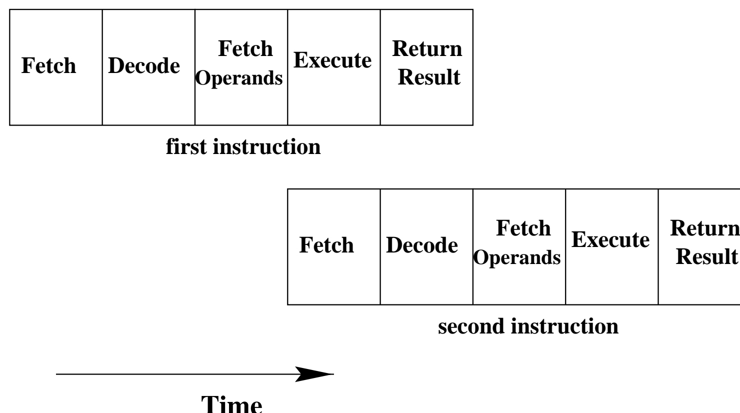
Always a *pipeline* of operations which must be performed sequentially.

The number of *stages* in the pipeline, or *pipeline depth*, can be between about 5 and 15 depending on the processor.

Making it go faster…

If each pipeline stage takes a single clock-cycle to complete, the previous scheme would suggest that it takes five clock cycles to execute a single instruction.

Clearly one can do better: in the absence of branch instructions, the next instruction can always be both fetched and decoded whilst the previous instruction is executing. This shortens our example to three clock cycles per instruction.



...and faster...

A functional unit may itself be pipelined. Considering again floating-point addition, even in base 10 there are three distinct stages to perform:

$9.67 \times 10^5 + 4 \times 10^4$

First the exponents are adjusted so that they are equal:

$9.67 \times 10^5 + 0.4 \times 10^5$

only then can the mantissas be added: 10.07×10^5

then one may have to readjust the exponent: 1.007×10^6

So floating point addition usually takes at least three clock cycles in the execution stage. But the adder may be able to start a new addition ever clock cycle, as these stages use distinct parts of the adder.

Such an adder would have a *latency* of three clock cycles, but a *repeat* or *issue rate* of one clock cycle.

...and faster...

Further improvements are governed by *data dependency*. Consider:

```
fadd f4,f5,f6
fmul f6,f7,f4
```

(Add `f4` and `f5` placing the result in `f6`, then multiply `f6` and `f7` placing the result back in `f4`.)

Clearly the add must finish (`f6` must be calculated) before the multiply can start. There is a data dependency between the multiply and the add.

But consider

```
fadd f4,f5,f6
fmul f3,f7,f9
```

Now any degree of overlap between these two instructions is permissible: they could even execute simultaneously or in the reverse order and still give the same result.

...and faster

We have now reached one instruction per cycle, assuming data independency.

If the instructions are short and simple, it is easy for the CPU to dispatch multiple instructions simultaneously, provided that each functional unit receives no more than one instruction per clock cycle.

So, in theory, an FP add, an FP multiply, an integer add, an FP load and an integer store might all be started simultaneously.

RISC instruction sets are carefully designed so that each instruction uses only one functional unit, and it is easy for the decode/issue logic to spot dependencies. CISC is a mess, with a single instruction potentially using several functional units.

CISC (Complex Instruction Set Computer) relies on a single instruction doing a lot of work: maybe incrementing a pointer and loading data from memory and doing an arithmetic operation.

RISC (Reduced Instruction Set Computer) relies on the instructions being very simple – the above CISC example would certainly be three RISC instructions – and then letting the CPU overlap them as much as possible.

Breaking Dependencies

```
for(i=0;i<n;i++){
  sum+=a[i];
}
```

```
do i=1,n
  sum=sum+a(i)
enddo
```

This would appear to require three clock cycles per iteration, as the iteration `sum=sum+a[i+1]` cannot start until `sum=sum+a[i]` has completed. However, consider

```
for(i=0;i<n;i+=3){
  s1+=a[i];
  s2+=a[i+1];
  s3+=a[i+2];
}
sum=s1+s2+s3;
```

```
do i=1,n,3
  s1=s1+a(i)
  s2=s2+a(i+1)
  s3=s3+a(i+2)
enddo
sum=s1+s2+s3
```

The three distinct partial sums have no interdependency, so one add can be issued every cycle.

Do not do this by hand. This is a job for an optimising compiler, as you need to know a lot about the particular processor you are using before you can tell how many paritial sums to use. And worrying about *codas* for `n` not divisible by 3 is tedious.

An Aside: Choices and Families

There are many choices to make in CPU design. Fixed length instructions, or variable? How many integer registers? How big? How many floating point registers (if any)? Should 'complicated' operations be supported? (Division, square roots, trig. functions, . . .). Should functional units have direct access to memory? Should instructions overwrite an argument with the result? Etc.

This has led to many different CPU families, with no compatibility existing between families, but backwards compatibility within families (newer members can run code compiled for older members).

In the past different families were common in desktop computers. Now the Intel/AMD family has a near monopoly here, but mobile phones usually contain ARM-based CPUs, and printers, routers, cameras etc., often contain MIPS-based CPUs. The Sony PlayStation uses CPUs derived from IBM's Power range, as do the Nintendo Wii and Microsoft Xbox.

At the other end of the computing scale, Intel/AMD has only recently begun to dominate. However, the top twenty machines in the November 2010 Top500 supercomputer list include three using the IBM Power series of processors, and another three using GPUs to assist performance. Back in June 2000, the Top500 list included a single Intel entry, admittedly top, the very specialised one-off ASCI Red. By June 2005 Intel's position had improved to 7 in the top 20.

Compilers

CPUs from different families will speak rather different languages, and, even within a family, new instructions get added from generation to generation to make use of new features.

Hence intelligent Humans write code in well-defined processor-independent languages, such as Fortran or C, and let the compiler do the work of producing the correct instructions for a given CPU. The compiler must also worry quite a lot about interfacing to a given operating system, so running a Windows executable on a machine running MacOS or Linux, even if they have the same CPU, is far from trivial (and generally impossible).

Compilers can, and do, of course, differ in how fast the sequence of instructions they translate code into runs, and even how accurate the translation is.

Well-defined processor-independent languages tend to be supported on by a wide variety of platforms over a long period of time. What I wrote a *long* time ago in Fortran 77 or ANSI C I can still run easily today. What I wrote in QuickBASIC then rewrote in TurboBASIC is now useless again, and became useless remarkably quickly.

Ignoring Intel

Despite Intel's desktop dominance, this course is utterly biased towards discussing RISC machines. It is not fun to explain an instruction such as
`faddl (%ecx,%eax,8)`
(add to the register at the top of the FP register stack the value found at the memory address given by the `ecx` register plus 8× the `eax` register) which uses an integer shift (×8), integer add, FP load and FP add in one instruction.

Furthermore, since the days of the Pentium Pro (1995), Intel's processors have had RISC cores, and a CISC to RISC translator feeding instructions to the core. The RISC core is never exposed to the programmer, leaving Intel free to change it dramatically between processors. A hideous operation like the above will be broken into three or four "μ-ops" for the core. A simpler CISC instruction might map to single μ-op (micro-op).

Designing a CISC core to do a decent degree of pipelining and simultaneous execution, when instructions may use multiple functional units, and memory operations are not neatly separated, is more painful than doing runtime CISC to RISC conversion.

A Branch in the Pipe

So far we have assumed a linear sequence of instructions. What happens if there is a branch?

```
   double t=0.0; int i,n;
   for (i=0;i<n;i++) t=t+x[i];

# $17 contains n, # $16 contains x
  fclr $f0
  clr $1
  ble $17,L$5
L$6:
  ldt   $f1, ($16)
  addl  $1, 1, $1
  cmplt $1, $17, $3
  lda   $16, 8($16)
  addt  $f0, $f1, $f0
  bne   $3, L$6
L$5:
```

```
t=0
do i=1,n
  t=t+x(i)
enddo
```

There will be a conditional jump or *branch* at the end of the loop. If the processor simply fetches and decodes the instructions following the branch, then when the branch is taken, the pipeline is suddenly empty.

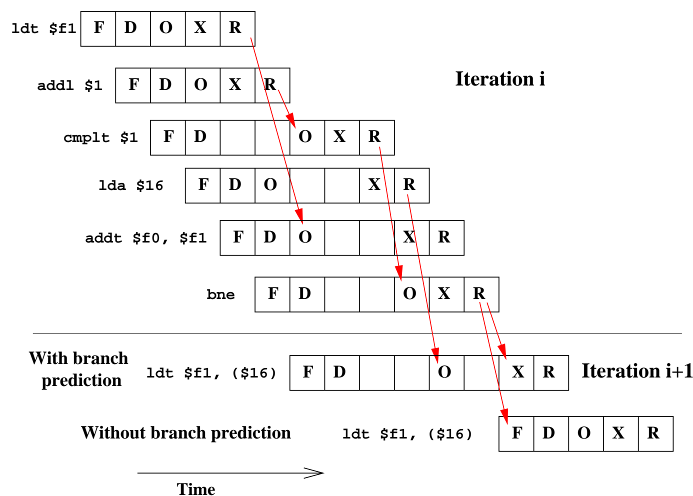
Assembler in More Detail

The above is Alpha assembler. The integer registers `$1`, `$3`, `$16` and `$17` are used, and the floating point registers `$f0` and `$f1`. The instructions are of the form ‘op a,b,c’ meaning ‘c=a op b’.

`fclr $f0`	Float CLeaR `$f0` – place zero in `$f0`
`clr $1`	CLeaR `$1`
`ble $17, L$5`	Branch if Less than or Equal on comparing `$17` to (an implicit) zero and jump to `L$5` if less (i.e. skip loop)
L$6:	
`ldt $f1, ($16)`	LoaD `$f1` with value value from memory from address `$16`
`addl $1, 1, $1`	$1=$1+1
`cmplt $1, $17, $3`	CoMPare `$1` to `$17` and place result in `$3`
`lda $16, 8($16)`	LoaD Address, effectively `$16=$16+8`
`addt $f0, $f1, $f0`	`$f0=$f0+$f1`
`bne $3,L$6`	Branch Not Equal – if counter ≠n, do another iteration
L$5:	

The above is only assembler anyway, readable by Humans. The machine-code instructions that the CPU actually interprets have a simple mapping from assembler, but will be different again. For the Alpha, each machine code instruction is four bytes long. For IA32 machines, between one and a dozen or so bytes.

Predictions



With the simplistic pipeline model of page 19, the loop will take 9 clock cycles per iteration if the CPU predicts the branch and fetches the next instruction appropriately. With no prediction, it will take 12 cycles.

A ‘real’ CPU has a pipeline *depth* much greater than the five slots shown here: usually ten to twenty. The penalty for a mispredicted branch is therefore large.

Note the *stalls* in the pipeline based on data dependencies (shown with red arrows) or to prevent the execution order changing. If the instruction fetch unit fetches one instruction per cycle, stalls will cause a build-up in the number of *in flight* instructions. Eventually the fetcher will pause to allow things to quieten down.

(This is not the correct timing for any Alpha processor.)

Speculation

In the above example, the CPU does not begin to execute the instruction after the branch until it knows whether the branch was taken: it merely fetches and decodes it, and collects its operands. A further level of sophistication allows the CPU to execute the next instruction(s), provided it is able to throw away all results and side-effects if the branch was mispredicted.

Such execution is called *speculative execution*. In the above example, it would enable the `ldt` to finish one cycle earlier, progressing to the point of writing to the register before the result of the branch were known.

More advanced forms of speculation would permit the write to the register to proceed, and would undo the write should the branch have been mispredicted.

Errors caused by speculated instructions must be carefully discarded. It is no use if

```
if (x>0) x=sqrt(x);
```

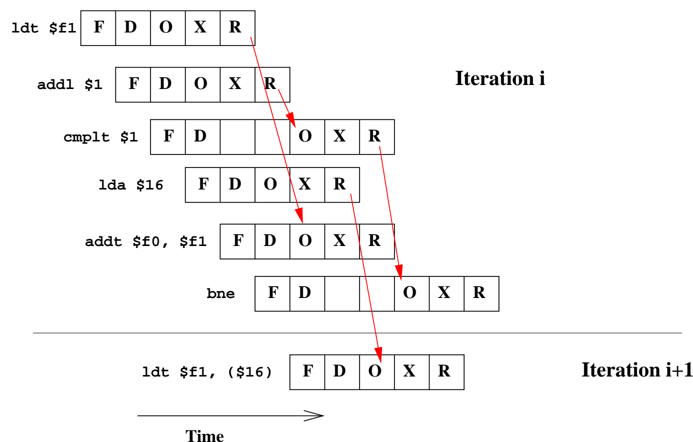
causes a crash when the square root is executed speculatively with `x=-1`, nor if

```
if (i<1000) x=a[i];
```

causes a crash when `i=2000` due to trying to access `a[2000]`.

Almost all current processors are capable of some degree of speculation.

OOO!



Previously the `cmplt` is delayed due to a dependency on the `addl` immediately preceeding it. However, the next instruction has no relevant dependencies. A processor capable of *out-of-order* execution could execute the `lda` before the `cmplt`.

The timing above assumes that the `ldt` of the next iteration can be executed speculatively and OOO before the branch. Different CPUs are capable of differing amounts of speculation and OOOE.

The EV6 Alpha does OOOE, the EV5 does not, nor does the UltraSPARC III. In this simple case, the compiler erred in not changing the order itself. However, the compiler was told not to optimise for this example.

Typical functional unit speeds

Instruction	Latency	Issue rate
iadd/isub	1	1
and, or, etc.	1	1
shift, rotate	1	1
load/store	1-2	1
imul	3-15	3-15
fadd	3	1
fmul	2-3	1
fdiv/fsqrt	15-25	15-25

In general, most things 1 to 3 clock cycles and pipelined, except integer $\times$ and $\div$, and floating point $\div$ and $\surd$.

'Typical' for simple RISC processors. Some processors tend to have longer fp latencies: 4 for `fadd` and `fmul` for the UltraSPARC III, 5 and 7 respectively for the Pentium 4, 3 and 5 respectively for the Core 2 / Nehalem / Sandy Bridge.

Floating Point Rules?

Those slow integer multiplies are more common that it would seem at first. Consider:

```
double precision x(1000),y(500,500)
```

The address of `x(i)` is the address of `x(1)` plus $8 \times (i-1)$. That multiplication is just a shift. However, `y(i,j)` is that of `y(1,1)` plus $8 \times ((i-1) + (j-1) \times 500)$. A lurking integer multiply!

Compilers may do quite a good job of eliminating unnecessary multiplies from common sequential access patterns.

C does things rather differently, but not necessarily better.

Hard or Soft?

The simple operations, such as $+$, $-$ and $*$ are performed by dedicated pipelined pieces of hardware which typically produce one result each clock cycle, and take around four clock cycles to produce a given result.

Slightly more complicated operations, such as $/$ and $\sqrt{}$ may be done with *microcode*. Microcode is a tiny program on the CPU itself which is executed when a particular instruction, e.g. $/$, is received, and which may use the other hardware units on the CPU multiple times.

Yet more difficult operations, such as trig. functions or logs, are usually done entirely with software in a library. The library uses a collection of power series or rational approximations to the function, and the CPU needs evaluate only the basic arithmetic operations.

The IA32 range is unusual in having microcoded instructions for trig. functions and logs. Even on a Core2 or Core i7, a single trig instruction can take over 100 clock cycles to execute. RISC CPUs tend to avoid microcode on this scale.

The trig. function instructions date from the old era of the x87 maths coprocessor, and no corresponding instruction exists for data in the newer SSE2/XMM registers.

Division by Multiplication?

There are many ways to perform floating point division. With a fast hardware multiplier, Newton-Raphson like iterative algorithms can be attractive.

$$x_{n+1} = 2x_n - b{x_n}^2$$

will, for reasonable starting guesses, converge to $1/b$. E.g., with $b = 6$.

n	x_n
0	0.2
1	0.16
2	0.1664
3	0.16666624
4	0.1666666666655744

How does one form an initial guess? Remember that the number is already stored as $m \times 2^e$, and $0.5 \leq m < 1$. So a guess of $0.75 \times 2^{1-e}$ is within a factor of 1.5. In practice the first few bits of m are used to index a lookup table to provide the initial guess of the mantissa.

A similar scheme enables one to find $1/\sqrt{b}$, and then $\sqrt{b} = b \times 1/\sqrt{b}$, using the recurrance $x \to 0.5x(3 - bx^2)$

Vector Computers

The phrase ‘vector computer’ means different things to different people.

To Cray, it meant having special ‘vector’ registers which store multiple floating point numbers, ‘multiple’ generally being 64, and on some models 128. These registers could be operated on using single instructions, which would perform element-wise operations on the whole register. Usually there would be just a single addition unit, but a vadd instruction would take around 70 clock cycles – one cycle per element, and a small pipeline start overhead.

So the idea was that the vector registers gave a simple mechanism for presenting a long sequence of independent operations to a highly pipelined functional unit.

Cray’s Other Idea

The other trick with vector Crays was to omit all data caches. Instead they employed a large number of banks of memory (typically sixteen), and used no-expense-spared SRAM for their main memory anyway. This gave them huge memory bandwidth. The Cray Y/MP-8, first sold in 1988, had a peak theoretical speed of 2.7 GFLOPS, about half that of a 2.8GHz Pentium 4 (introduced 2002). However, its memory bandwidth of around 27GB/s was considerably better than the P4 at under 4GB/s, and would still beat a single-socket Ivy Bridge machine (introduced 2012).

Not only was the Cray’s ratio of memory bandwidth to floating point performance about fifty times higher than a current desktop, but its memory latency was low – lower than that of a current desktop machine – despite its clock speed being only 167MHz. The memory controller on the Cray could handle many outstanding memory requests, which further hid latencies. Of course, memory requests were likely to be generated 64 words at a time.

The weak spot of this system was dealing with strides which contained a large power of two. A stride of 16 doubles might cause all requests to go to a single memory bank, reducing performance by at least a factor of ten.

Intel and Vectorisation

Intel's approach to vectorisation is very different from Cray's. The memory architecture is unchanged. The vector length is two or four (eight for the Xeon Phi), not sixty four.

But, Intel provides sufficient functional units, as putting extra transistors on a chip is now cheap, to operate on a whole vector at once. The vectorisation is used to keep multiple functional units busy at once, not in order to make efficient use of the pipeline in a single functional unit.

A Cray will run reasonably fast streaming huge arrays from main memory, with 10 bytes of memory bandwidth per peak FLOPS. An Intel processor, with around 0.2 bytes of memory bandwidth, won't. One needs to worry about its caches to get decent performance.

Meaningless Indicators of Performance

The only relevant performance indicator is how long a computer takes to run *your* code. Thus my fastest computer is not necessarily your fastest computer.

Often one buys a computer before one writes the code it has been bought for, so other 'real-world' metrics are useful. Some are not useful:

- MHz: the silliest: some CPUs take 4 clock cycles to perform one operation, others perform four operations in one clock cycle. Only any use when comparing otherwise identical CPUs.
- MIPS: Millions of Instructions Per Second. Theoretical peak speed of decode/issue logic, or maybe the time taken to run a 1970's benchmark. Gave rise to the name Meaningless Indicator of Performance.
- FLOPS: Floating Point Operations Per Second. Theoretical peak issue rate for floating point computational instructions, ignoring loads and stores and with optimal ratio of $+$ to $*$. Hence MFLOPS, GFLOPS, TFLOPS: 10^6, 10^9, 10^{12} FLOPS.

The Guilty Candidates: Linpack

Linpack 100x100

Solve 100x100 set of double precision linear equations using fixed FORTRAN source. Pity it takes just 0.7 s at 1 MFLOPS and uses under 100KB of memory. Only relevant for pocket calculators.

Linpack 1000x1000 or nxn

Solve 1000x1000 (or nxn) set of double precision linear equations by any means. Usually coded using a blocking method, often in assembler. Is that relevant to your style of coding? Achieving less than 50% of a processor's theoretical peak performance is unusual.

Linpack is convenient in that it has an equal number of adds and multiplies uniformly distributed throughout the code. Thus a CPU with an equal number of FP adders and multipliers, and the ability to issue instructions to all simultaneously, can keep all busy.

Number of operations: O(n^3), memory usage O(n^2).
n chosen by manufacturer to maximise performance, which is reported in GFLOPS.

SPEC

SPEC is a non-profit benchmarking organisation. It has two CPU benchmarking suites, one concentrating on integer performance, and one on floating point. Each consists of around ten programs, and the mean performance is reported.

Unfortunately, the benchmark suites need constant revision to keep ahead of CPU developments. The first was released in 1989, the second in 1992, the third in 1995. None of these use more than 8MB of data, so fit in cache with many current computers. Hence a fourth suite was released in 2000, and then another in 2006.

It is not possible to compare results from one suite with those from another, and the source is not publically available.

Until 2000, the floating point suite was entirely Fortran.

Two scores are reported, 'base', which permits two optimisation flags to the compiler, and 'peak' which allows any number of compiler flags. Changing the code is not permitted.

SPEC: Standard Performance Evaluation Corporation (`www.spec.org`)

SPEC rates

SPEC also has a set of throughput benchmarks, which consist of running multiple copies of their serial benchmarks simultaneously. For multicore machines, this should work well, only in practice the cores compete for limited memory bandwidth, and it works less well than one might hope.

For instance, in late 2008 Intel published a result of 155 for a 24 core X7460 system. This essentially has four six-core Core 2 processors running at 2.66GHz. Clearly much faster than a single dual-core Core 2 processor at the same clock speed. However, the Core 2 E8200 achieved a score of 28.9 on this benchmark over six months earlier, so twelve times the core count has resulted in less than six times the throughput running serial codes.

At that time, AMD was easily beating Intel with four-socket machines. With four quad-core Opterons running at just 2.3GHz it could match the performance of the 24 core Intel machine, and at 2.7GHz it could achieve just over 200.

Intel has since caught up with AMD, and by early 2011 Intel could manage a score of 1150 using eight 8-core Xeon X7560s at 2.27GHz, whereas AMD scored 1310 with eight 12-core Opterons at 2.5GHz. The biggest machine to run this benchmark was an IBM with 32 8-core Power7 processors running at 4GHz, which scored 10,500.

Your Benchmark or Mine?

Picking the oldest desktop in TCM, a 2.67GHz Pentium 4, and the newest, a 3.1GHz quad core ‘Haswell’ CPU, I ran two benchmarks.

Linpack gave results of 3.88 GFLOPS for the P4, and 135 GFLOPS for the Haswell, a win for the Haswell by a factor of around 35.

A nasty synthetic integer benchmark I wrote gave run-times of 6.0s on the P4, and 9.7s on the Haswell, a win for the P4 by a factor of 1.6 in speed.

(Linux’s notoriously meaningless ‘BogoMIPS’ benchmark is slightly kinder to the Haswell, giving it a score of 6,185 against 5,350 for the P4.)

It is all too easy for a vendor to use the benchmark of his choice to prove that his computer is faster than a given competitor.

The P4 was a ‘Northwood’ P4 first sold in 2002, the Haswell was first sold in 2013.

Memory

- DRAM
- Parity and ECC
- Going faster: wide bursts
- Going faster: caches

Memory Design

The first DRAM cell requiring just one transistor and one capacitor to store one bit was invented and produced by Intel in 1974. It was mostly responsible for the early importance of Intel as a chip manufacturer.

The design of DRAM has changed little. The speed, as we shall soon see, has changed little. The price has changed enormously. I can remember when memory cost around £1 per KB (early 1980s). It now costs around 1p per MB, a change of a factor of 10^5, or a little more in real terms. This change in price has allowed a dramatic change in the amount of memory which a computer typically has.

Alternatives to DRAM are SRAM – very fast, but needs six transitors per bit, and flash RAM – unique in retaining data in the absence of power, but writes are slow *and* cause significant wear.

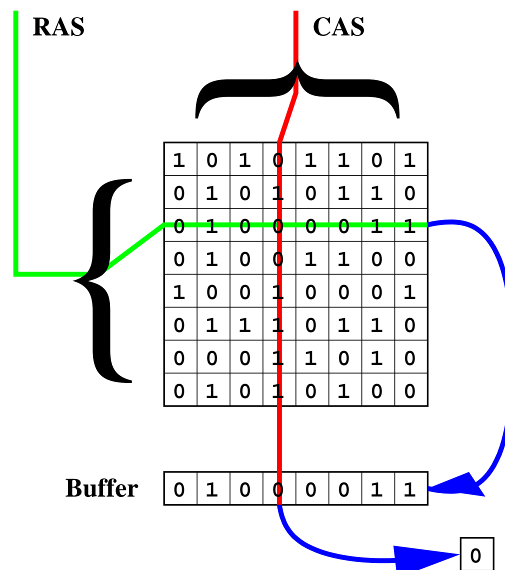
RAM: Random Access Memory – i.e. not block access (disk drive), nor sequential access (tape drive).

D-RAM

The charge in a DRAM cell slowly leaks away. So each cell is read, and then written back to, several times a second by *refresh* circuitary to keep the contents stable. This is why this type of memory is called Dynamic RAM.

Of course, as anyone in the HEP community can testify, one can charge lots of small capacitors, monitor their charge, and a sudden change means an ionisation event has occured in their dielectric – an energetic particle has been detected. DRAM is worrying similar to a semiconductor particle detector, so cautious people use extra DRAM cells to store an error correction / detection code so that stray cosmic rays do not end up in one's results. Such memory is called ECC memory. (Error Correcting Code.)

DRAM in Detail



DRAM cells are arranged in (near-)square arrays. To read, first a row is selected and copied to a buffer, from which a column is selected, and the resulting single bit becomes the output. This example is a 64 bit DRAM.

This chip would need 3 *address lines* (i.e. pins) allowing 3 bits of address data to be presented at once, and a single `data line`. Also two pins for power, two for CAS and RAS, and one to indicate whether a read or a write is required.

Of course a 'real' DRAM chip would contain several tens of million bits.

DRAM Read Timings

To read a single bit from a DRAM chip, the following sequence takes place:

- Row placed on address lines, and Row Access Strobe pin signalled.
- After a suitable delay, column placed on address lines, and Column Access Strobe pin signalled.
- After another delay the one bit is ready for collection.
- The DRAM chip will automatically write the row back again, and will not accept a new row address until it has done so.

The same address lines are used for both the row and column access. This halves the number of addess lines needed, and adds the RAS and CAS pins.

Reading a DRAM cell causes a significant drain in the charge on its capacitor, so it needs to be refreshed before being read again.

More Speed!

The above procedure is tediously slow. However, for reading consecutive addresses, one important improvement can be made.

Having copied a whole row into the buffer (which is usually SRAM (see later)), if another bit from the same row is required, simply changing the column address whilst signalling the CAS pin is sufficient. There is no need to wait for the chip to write the row back, and then to rerequest the same row. Thus Fast Page Mode (FPM) and Extended Data Out (EDO) DRAM.

Today's SDRAM (Synchronous DRAM) takes this approach one stage further. It assumes that the next (several) bits are wanted, and sends them in sequence without waiting to receive requests for their column addresses.

Speed

Old-style memory quoted latencies which were simply the time it would take an idle chip to respond to a memory request. In the early 1980s this was about 250ns. By the early 1990s it was about 80ns.

Today timings are quoted as clock cycles for column access to data out (T_{CL} or T_{CAS}) and idle to row select finished (T_{RCD}) These are the first two numbers of the four usually quoted for memory timings. The clock refered to is the undoubled data clock, so a DDR3-1333 module with timings of 7-7-7-24 has a latency of 14 cycles of a 667MHz clock, or 21ns.

So in twenty years memory has got four times faster in terms of latency.

Bandwidth

Bandwidth has improved much more over the same period. In the early 1980s memory was usually arranged to deliver 8 bits (one byte) at once, with eight chips working in parallel. By the early 1990s that had risen to 32 bits (4 bytes), and today one expects 128 bits (16 bytes) on any desktop.

More dramatic is the change in time taken to access consecutive items. In the 1980s the next item (whatever it was) took slightly longer to access, for the DRAM chip needed time to recover from the previous operation. So late 1980s 32 bit wide 80ns memory was unlikely to deliver as much as four bytes every 100ns, or 40MB/s. Now sequential access is anticipated, and arrives at the doubled clock speed, so at 1333MHz for DDR3-1333 memory. Coupled with being arranged with 128 bits in parallel, this leads to a theoretical bandwidth of 20GB/s.

So in twenty years the bandwidth has improved by a factor of about 500.

Parity

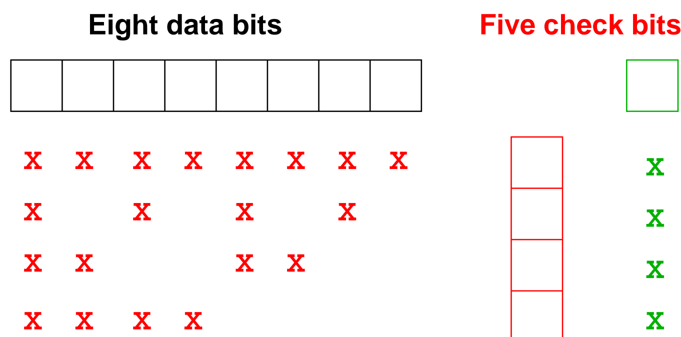
In the 1980s, business computers had memory arranged in bytes, with one extra bit per byte which stored parity information. This is simply the sum of the other bits, modulo 2. If the parity bit disagreed with the contents of the other eight bits, then the memory had suffered physical corruption, and the computer would usually crash, which is considered better than calmly going on generating wrong answers.

Calculating the parity value is quite cheap in terms of speed and complexity, and the extra storage needed is only 12.5%. However parity will detect only an odd number of bit-flips in the data protected by each parity bit. If an even number change, it does not notice. And it can never correct.

ECC

Better than parity is ECC memory (Error Correcting Code), usually SEC-DED (Single Error Corrected, Double Error Detected).

One code for dealing with n bits requires an extra $2+\log_2 n$ check bits. Each code now usually protects eight bytes, 64 bits, for which $2+\log_2 64 = 8$ extra check bits are needed. Once more, 12.5% extra, or one extra bit per byte. The example shows an ECC code operating on 8 bits of data.



One check bit is a parity check of the other check bits (green, top right), else errors in the check bits are undetected and cause erroneous 'corrections'. The other four check bits (red column) store parity information for the data bits indicated. A failing data bit causes a unique pattern in these bits. This is not the precise code used, & fails to detect 2-bit errors, but it shows the general principle.

Computers with parity could detect one bit in error per byte. Today's usual ECC code can correct a one bit error per 8 bytes, and detect a two bit error per eight bytes. Look up Hamming Codes for more information.

Causes and Prevelance of Errors

In the past most DRAM errors have been blamed on cosmic rays, more recent research suggest that this is not so. A study of Google's servers over a 30 month period suggests that faulty chips are a greater problem. Cosmic rays would be uniformly distributed, but the errors were much more clustered.

About 30% of the servers had at least one correctable error per year, but the average number of correctable errors per machine year was over 22,000. The probability of a machine which had one error having another within a year was 93%. The uncorrectable error rate was 1.3% per machine year.

The numbers are skewed by the fact that once insulation fails so as to lock a bit to one (or zero), then, on average, half the accesses will result in errors. In practice insulation can partially fail, such that the data are usually correct, unless neighbouring bits, temperature, . . . , conspire to cause trouble.

Uncorrectable errors were usually preceded by correctable ones: over 60% of uncorrectable errors had been preceded by a correctable error in the same DIMM in the same month, whereas a random DIMM has a less than 1% correctable error rate per month.

'DRAM Errors in the Wild: a Large-Scale Field Study', Schroeder *et al.*

ECC: Do We Care?

A typical home PC, run for a few hours each day, with only about half as much memory as those Google servers, is unlikely to see an error in its five year life. One has about a one in ten chance of being unlucky. When running a hundred machines 24/7, the chances of getting through a month, let alone a year, without a correctable error would seem to be low.

Intel's desktop i3/i5/i7 processors do not support ECC memory, whereas their server-class Xeon processors all do. Most major server manufacturers (HP, Dell, IBM, etc.) simply do not sell any servers without ECC. Indeed, most also support the more sophisticated 'Chipkill' correction which can cope with one whole chip failing on a bus of 128 data bits and 16 'parity' bits.

I have an 'ECC only' policy for servers, both file servers and machines likely to run jobs. In my Group, this means every desktop machine. The idea of doing financial calculations on a machine without ECC I find amusing and unauditable, but I realise that, in practice, it is what most Accounts Offices do. But money matters less than science.

Of course an undetected error may cause an immediate crash, it may cause results to be obviously wrong, it may cause results to be subtly wrong, or it may have no impact on the final result.

'Chipkill' is IBM's trademark for a technology which Intel calls Intel x4 SDDC (single device data correction). It starts by interleaving the bits to form four 36 bit words, each word having one bit from each chip, so a SEC-DED code is sufficient for each word.

Keeping up with the CPU

CPU clock speeds in the past twenty years have increased by a factor of around 500. (About 60MHz to about 3GHz.) Their performance in terms of instructions per second has increased by about 10,000, as now one generally has four cores, each capable of multiple instructions per clock cycle, not a single core struggling to maintain one instruction per clock cycle.

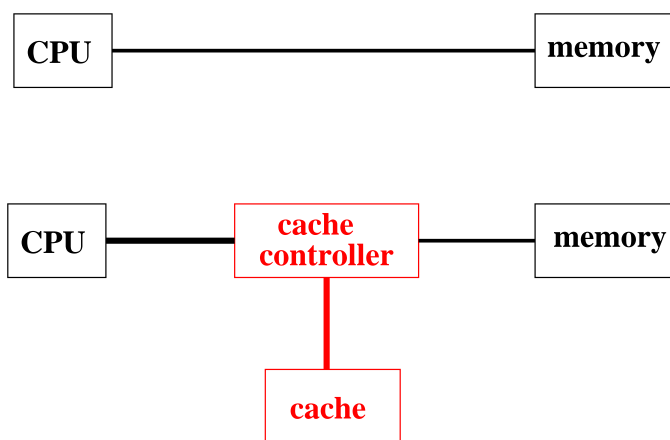
The partial answer is to use expensive, fast, cache RAM to store frequently accessed data. Cache is expensive because its SRAM uses multiple transistors per bit (typically six). It is fast, with sub-ns latency, lacking the output buffer of DRAM, and not penalising random access patterns.

But it is power-hungry, space-hungry, and needs to be physically very close to the CPU so that distance does not cause delay. $c = 1$ in units of feet per ns in vacuum. So a 3GHz signal which needs to travel just two inches and back again will lose a complete cycle. In silicon things are worse.

(Experimentalists claim that $c = 0.984$ft/ns.)

Caches: the Theory

The theory of caching is very simple. Put a small amount of fast, expensive memory in a computer, and arrange automatically for that memory to store the data which are accessed frequently. One can then define a cache *hit rate*, that is, the number of memory accesses which go to the cache divided by the total number of memory accesses. This is usually expressed as a percentage & will depend on the code run.



The first paper to describe caches was published in 1965 by Maurice Wilkes (Cambridge). The first commercial computer to use a cache was the IBM 360/85 in 1968.

The Cache Controller

Conceptually this has a simple task:

- Intercept every memory request
- Determine whether cache holds requested data
- If so, read data from cache
- If not, read data from memory *and* place a copy in the cache as it goes past.

However, the second bullet point must be done *very* fast, and this leads to the compromises. A cache controller inevitably makes misses slower than they would have been in the absence of any cache, so to show a net speed-up hits have to be plentiful and fast. A badly designed cache controller can be worse than no cache at all.

An aside: Hex

A quick lesson in hexadecimal (base-16) arithmetic is due at this point. Computers use base-2, but humans tend not to like reading long base-2 numbers.

Humans also object to converting between base-2 and base-10.

However, getting humans to work in base-16 and convert between base-2 and base-16 is easier.

Hex uses the letters A to F to represent the 'digits' 10 to 15. As $2^4 = 16$ conversion to and from binary is done trivially using groups of four digits.

Converting to / from Hex

```
0101 1101 0010 1010 1111 0001 1100 0011

  5    C    2    A    F    1    B    3
```

So

$$0101\,1101\,0010\,1010\,1111\,0001\,1100\,0011_2$$

$$= 5\text{C}2\text{A}\,\text{F}1\text{B}3_{16} = 1,546,318,259$$

As one hex digit is equivalent to four binary digits, two hex digits are exactly sufficient for one byte.

Hex numbers are often prefixed with '`0x`' to distinguish them from base ten.

When forced to work in binary, it is usual to group the digits in fours as above, for easy conversion into hex or bytes.

Our Computer

For the purposes of considering caches, let us consider a computer with a 1MB address space and a 64KB cache.

An address is therefore 20 bits long, or 5 hex digits, or $2\frac{1}{2}$ bytes.

Suppose we try to cache individual bytes. Each entry in the cache must store not only the data, but also the address in main memory it was taken from, called the *tag*. That way, the cache controller can look through all the tags and determine whether a particular byte is in the cache or not.

So we have 65536 single byte entries, each with a $2\frac{1}{2}$ byte tag.

A Disaster

This is bad on two counts.

A waste of space

We have 64KB of cache storing useful data, and 160KB storing tags.

A waste of time

We need to scan 65536 tags before we know whether something is in the cache or not. This will take far too long.

Lines

The solution to the space problem is not to track bytes, but *lines*. Consider a cache which deals in units of 16 bytes.

```
64KB = 65536 * 1 byte
     =  4096 * 16 bytes
```

We now need just 4096 tags.

Furthermore, each tag can be shorter. Consider a random address:

```
0x23D17
```

This can be read as byte `7` of line `23D1`. The cache will either have all of line `23D1` and be able to return byte number 7, or it will have none of it. Lines always start at an address which is a multiple of their length.

Getting better…

A waste of space?

We now have 64KB storing useful data, and 8KB storing tags. Considerably better.

A waste of time

Scanning 4096 tags may be a 16-fold improvement, but is still a disaster.

Causing trouble

Because the cache can store only full lines, if the processor requests a single byte which the cache does not hold, the cache then requests the full line from the memory so that it can keep a copy of the line. Thus the memory might have to supply 16× as much data as before!

A Further Compromise

We have 4096 lines, potentially addressable as line 0 to line 0xFFF.

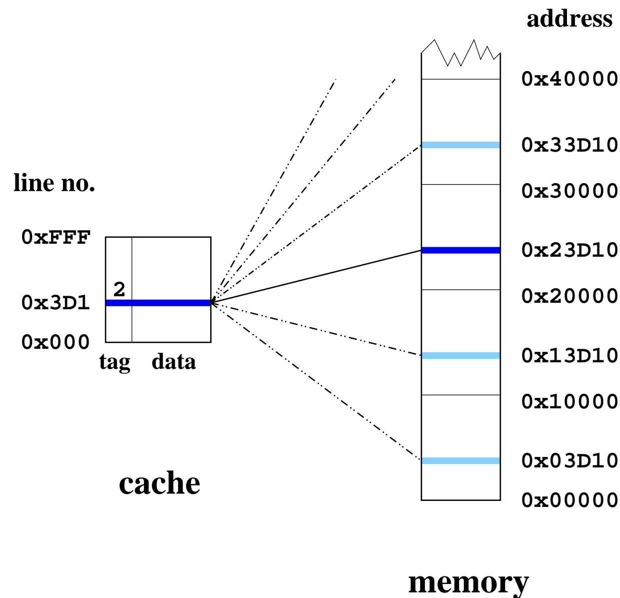
On seeing an address, e.g. `0x23D17`, we discard the last 4 bits, and scan all 4096 tags for the number `0x23D1`.

Why not always use line number `0x3D1` within the cache for storing this bit of memory? The advantage is clear: we need only look at one tag, and see if it holds the line we want, `0x23D1`, or one of the other 15 it could hold: `0x03D1, 0x13D1`, etc.

Indeed, the new-style tag need only hold that first hex digit, we know the other digits! This reduces the amount of tag memory to 2KB.

Direct Mapped Caches

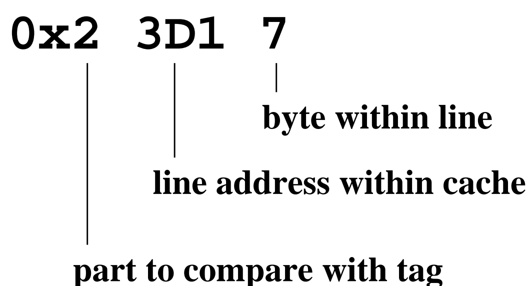
We have just developed a *direct mapped* cache. Each address in memory maps directly to a single location in cache, and each location in cache maps to multiple (here 16) locations in memory.



Success?

- The overhead for storing tags is 3%. Quite acceptable, and much better than 250%!
- Each 'hit' requires a tag to be looked up, a comparison to be made, and then the data to be fetched. Oh dear. This *tag RAM* had better be *very* fast.
- Each miss requires a tag to be looked up, a comparison to fail, and then a whole line to be fetched from main memory.
- The 'decoding' of an address into its various parts is instantaneous.

The zero-effort address decoding is an important feature of all cache schemes.



The Consequences of Compromise

At first glance we have done quite well. Any contiguous 64KB region of memory can be held in cache. (As long as it starts on a cache line boundary)

E.g. The 64KB region from `0x23840` to `0x3383F` would be held in cache lines `0x384` to `0xFFF` then `0x000` to `0x383`

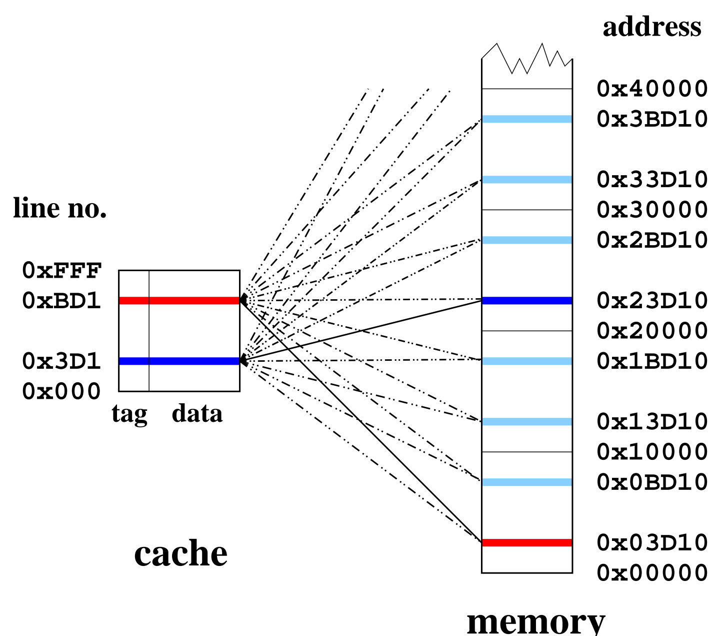
Even better, widely separated pieces of memory can be in cache simultaneously. E.g. `0x15674` in line `0x567` and `0xC4288` in line `0x428`.

However, consider trying to cache the two bytes `0x03D11` and `0x23D19`. This cannot be done: both map to line `0x3D1` within the cache, but one requires the memory area from `0x03D10` to be held there, the other the area from `0x23D10`.

Repeated accesses to these two bytes would cause cache *thrashing*, as the cache repeatedly caches then throws out the same two pieces of data.

Associativity

Rather than each line in memory being storable in just one location in cache, why not make it two?



Thus a *2-way associative* cache, which requires two tags to be inspected for every access & an extra bit per tag. Can generalise to 2^n-way associativity.

Anti Thrashing Entries

Anti Thrashing Entries are a cheap way of increasing the effective associativity of a cache for simple cases. One extra cache line, complete with tag, is stored, and it contains the last line expelled from the cache proper.

This line is checked for a 'hit' in parallel with the rest of the cache, and if a hit occurs, it is moved back into the main cache, and the line it replaces is moved into the ATE.

Some caches have several ATEs, rather than just one.

```
double precision a(2048,2),x
do i=1,2048
  x=x+a(i,1)*a(i,2)
enddo
```

```
double a[2][2048],x;
for(i=0;i<2047;i++){
  x+=a[0][i]*a[1][i];
}
```

Assume a 16K direct mapped cache with 32 byte lines. `a(1,1)` comes into cache, pulling `a(2-4,1)` with it. Then `a(1,2)` displaces all these, at it must be stored in the same line, as its address modulo 16K is the same. So a(2,1) is not found in cache when it is referenced. With a single ATE, the cache hit rate jumps from 0% to 75%, the same that a 2-way associative cache would have achieved for this algorithm.

Remember that Fortran and C store arrays in the opposite order in memory. Fortran will have `a(1,1)`, `a(2,1)`, `a(3,1)...`, whereas C will have `a[0][0]`, `a[0][1]`, `a[0][2]...`

A Hierarchy

The speed gap between main memory and the CPU core is so great that there are usually multiple levels of cache.

The first level, or *primary cache*, is small (typically 16KB to 128KB), physically attached to the CPU, and runs as fast as possible.

The next level, or *secondary cache*, is larger (typically 256KB to 8MB), slower, and has a higher associativity. There may even be a third level too.

Typical times in clock-cycles to serve a memory request would be:

primary cache	2-4
secondary cache	5-25
main memory	30-300

Cf. functional unit speeds on page 32.

Intel tends to make small, fast caches, compared to RISC workstations which tend to have larger, slower caches.

Write Back or Write Through?

Should data written by the CPU modify merely the cache if those data are currently held in cache, or modify the memory too? The former, *write back*, can be faster, but the latter, *write through*, is simpler.

With a write through cache, the definitive copy of data is in the main memory. If something other than the CPU (e.g. a disk controller or a second CPU) writes directly to memory, the cache controller must *snoop* this traffic, and, if it also has those data in its cache, update (or invalidate) the cache line too.

Write back caches add two problems. Firstly, anything else reading directly from main memory must have its read intercepted if the cached data for that address differ from the data in main memory.

Secondly, on ejecting an old line from the cache to make room for a new one, if the old line has been modified it must first be written back to memory.

Each cache line therefore has an extra bit in its tag, which records whether the line is modified, or *dirty*.

Cache Design Decision

If a write is a miss, should the cache line be filled (as it would for a read)? If the data just written are read again soon afterwards, filling is beneficial, as it is if a write to the same line is about to occur. However, caches which allocate on writes perform badly on randomly scattered writes. Each write of one word is converted into *reading* the cache line from memory, modifying the word written in cache and marking the whole line dirty. When the line needs discarding, the whole line will be written to memory. Thus writing one word has been turned into two lines worth of memory traffic.

What line size should be used? What associativity?

If a cache is n-way associative, which of the n possible lines should be discarded to make way for a new line? A random line? The least recently used? A random line excluding the most recently used?

As should now be clear, not all caches are equal!

The 'random line excluding the most recently used' replacement algorithm (also called pseudo-LRU) is easy to implement. One bit marks the most recently used line of the associative set. True LRU is harder (except for 2-way associative).

Not All Data are Equal

If the cache controller is closely associated with the CPU, it can distinguish memory requests from the instruction fetcher from those from the load/store units. Thus instructions and data can be cached separately.

This almost universal *Harvard Architecture* prevents poor data access patterns leaving both data and program uncached. However, usually only the first level of cache is split in this fashion.

The instruction cache is usually write-through, whereas the data cache is usually write-back. Write-through caches never contain the 'master' copy of any data, so they can be protected by simple parity bits, and the master copy reloaded on error. Write back caches ought to be protected by some form of ECC, for if they suffer an error, they may have the only copy of the data now corrupted.

The term 'Harvard architecture' comes from an early American computer which used physically separate areas of main memory for storing data and instructions. No modern computer does this.

Explicit Prefetching

One spin-off from caching is the possibility of *prefetching*.

Many processors have an instruction which requests that data be moved from main memory to primary cache when it is next convenient.

If such an instruction is issued ahead of some data being required by the CPU core, then the data may have been moved to the primary cache by the time the CPU core actually wants them. If so, much faster access results. If not, it doesn't matter.

If the latency to main memory is 100 clock cycles, the prefetch instruction ideally needs issuing 100 cycles in advance, and many tens of prefetches might be busily fetching simultaneously. Most current processors can handle a couple of simultaneous prefetches...

Implicit Prefetching

Some memory controllers are capable of spotting certain access patterns as a program runs, and prefetching data automatically. Such prefetching is often called *streaming*.

The degree to which patterns can be spotted varies. Unit stride is easy, as is unit stride backwards. Spotting different simultaneous streams is also essential, as a simple dot product:

```
do i=1,n
  d=d+a(i)*b(i)
enddo
```

leads to alternate unit-stride accesses for `a` and `b`.

IBM's Power3 processor, and Intel's Pentium 4 both spotted simple patterns in this way. Unlike software prefetching, no support from the compiler is required, and no instructions exist to make the code larger and occupy the instruction decoder. However, streaming is less flexible.

Clock multiplying

Today all of the caches are usually found on the CPU die, rather than on external chips. Whilst the CPU is achieving hits on its caches, it is unaffected by the slow speed of the outside world (e.g. main memory).

Thus it makes sense for the CPU internally to use much higher clock-speeds than its external bus. The gap is actually decreasing currently as CPU speeds are levelling off at around 3GHz, whereas external bus speeds are continuing to rise. In former days the gap could be very large, such as the last of the Pentium IIIs which ran at around 1GHz internally, with a 133MHz external bus. In the days when caches were external to the CPU on the motherboard there was very little point in the CPU running faster than its bus. Now it works well provided that the cache hit rate is high (>90%), which will depend on both the cache architecture and the program being run.

In order to reduce power usage, not all of the CPU die uses the same clock frequency. It is common for the last level cache, which is responsible for around half the area of the die, to use clock speeds which are only around a half or a third of those of the CPU core and the primary cache.

Thermal Limits to Clock Multiplying

The rate at which the transistors which make up a CPU switch is controlled by the rate at which carriers get driven out of their gate regions. For a given chip, increasing the electric field, i.e. increasing the voltage, will increase this speed. Until the voltage is so high that the insulation fails.

The heat generated by a CPU contains both a simple ohmic term, proportional to the square of the voltage, and a term from the charging of capacitors through a resistor (modelling the change in state of data lines and transistors). This is proportional to both frequency and the square of the voltage.

Once the CPU gets too hot, thermally excited carriers begin to swamp the intrinsic carriers introduced by the n and p doping. With the low band-gap of silicon, the maximum junction temperature is around 90°C, or just 50°C above the air temperature which most computers can allegedly survive.

Current techniques allow around 120W to be dissipated from a chip with forced air cooling.

Laptops, and the more modern desktops, have power-saving modes in which the clock speed is first dropped, and then a fraction of a second later, the supply voltage also dropped.

The Relevance of Theory

```
integer a(*),i,j

j=1
do i=1,n
  j=a(j)
enddo
```

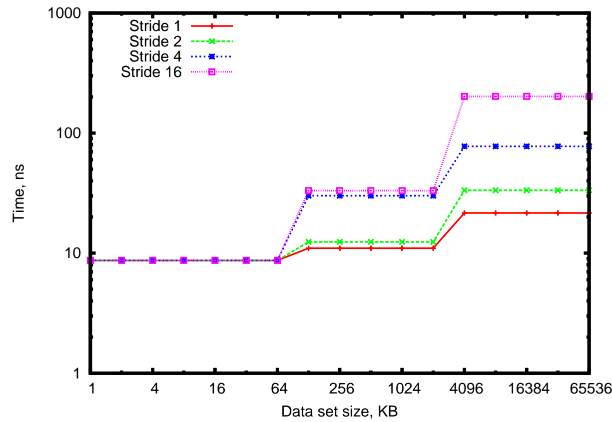
```
int i,j,*a;

j=1;
for (i=0;i<n;i++){
  j=a[j];
}
```

This code is mad. Every iteration depends on the previous one, and significant optimisation is impossible.

However, the memory access pattern can be changed dramatically by changing the contents of `a`. Setting `a(i)=i+1` and `a(k)=1` will give consecutive accesses repeating over the first `k` elements, whereas `a(i)=i+2`, `a(k-1)=2` and `a(k)=1` will access alternate elements, etc.

One can also try pseudorandom access patterns. They tend to be as bad as large stride access.

Classic caches

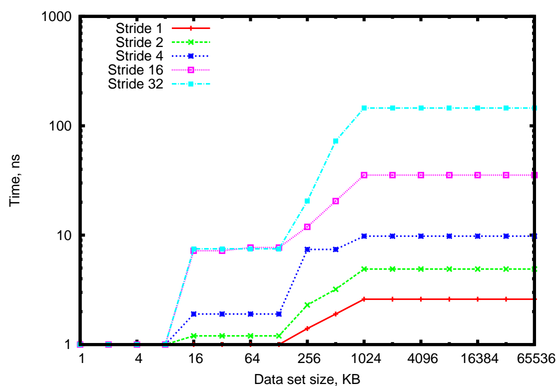


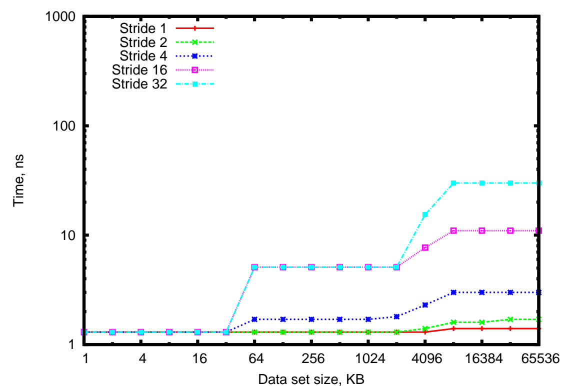
With a 16 element (64 bytes) stride, we see access times of 8.7ns for primary cache, 33ns for secondary, and 202ns for main memory. The cache sizes are clearly 64KB and 2MB.

With a 1 element (4 bytes) stride, the secondary cache and main memory appear to be faster. This is because once a cache line has been fetched from memory, the next 15 accesses will be primary cache hits on the next elements of that line. The average should be $(15 * 8.7 + 202)/16 = 20.7$ns, and 21.6ns is observed.

The computer used for this was a 463MHz XP900 (Alpha 21264). It has 64 byte cache lines.

Performance Enhancement





On the left a 2.4GHz Pentium 4 (launched 2002, RAMBUS memory), and on the right a 2.4GHz Core 2 quad core (launched 2008, DDR3 memory). Both have 64 byte cache lines.

For the Pentium 4, the fast 8KB primary cache is clearly seen, and a 512KB secondary less clearly so. The factor of four difference between the main memory's latency at a 64 byte and 128 byte stride is caused by automatic hardware prefetching into the secondary cache. For strides of up to 64 bytes inclusive, the hardware notices the memory access pattern, even though it is hidden at the software level, and starts fetching data in advance automatically.

For the Core 2 the caches are larger – 32KB and 4MB, and the main memory is a little faster. But six years and three generations of memory technology have changed remarkably little.

Matrix Multiplication: $A_{ij} = B_{ik}C_{kj}$

```
do i=1,n
  do j=1,n
    t=0
    do k=1,n
      t=t+b(i,k)*c(k,j)
    enddo
    a(i,j)=t
  enddo
enddo
```

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    t=0;
    for(k=0;k<n;k++){
      t+=b[i][k]*c[k][j];
    }
    a[i][j]=t;
  }
}
```

The above Fortran has unit stride access on the array `c` in the inner loop, but a stride of `n` doubles on the array `b`. The C manages unit stride on `b` and a stride of `n` doubles on the array `c`. Neither manages unit stride on both arrays.

Optimising this is not completely trivial, but is very worthwhile.

Very Worthwhile

The above code running on a 2.4GHz Core 2 managed around 500 MFLOPS at a matrix size of 64, dropping to 115 MFLOPS for a matrix size of 1024.

Using an optimised linear algebra library increased the speed for the smaller sizes to around 4,000 MFLOPS, and for the larger sizes to around 8,700 MFLOPS, close to the computer's peak speed of 9,600 MFLOPS.

There are many possibilities to consider for optimising this code. If the matrix size is very small, don't, for it will all fit in L1 cache anyway. For large matrices one can consider transposing the matrix which would otherwise be accessed with the large stride. This is most beneficial if that matrix can then be discarded (or, better, generated in the transposed form). Otherwise one tries to modify the access pattern with tricks such as

```
do i=1,nn,2
  do j=1,nn
    t1=0 ; t2=0
    do k=1,nn
      t1=t1+b(i,k)*c(k,j)        ! Remember that b(i,k) and
      t2=t2+b(i+1,k)*c(k,j)      ! b(i+1,k) are adjacent in memory
    enddo
    a(i,j)=t1
    a(i+1,j)=t2
  enddo
enddo
```

This halves the number of passes through `b` with the large stride, and therefore shows an immediate doubling of speed at `n=1024` from 115 MFLOPS to 230 MFLOPS. Much more to be done before one reaches 8,000 MFLOPS though, so don't bother: link with a good BLAS library and use its matrix multiplication routine! (Or use the F90 intrinsic `matmul` function in this case.) [If trying this at home, note that many Fortran compilers spot simple examples of matrix multiplication and re-arrange the loops themselves. This can cause confusion.]

Memory Access Patterns in Practice

Matrix Multiplication

We have just seen that very different speeds of execution can be obtained by different methods of matrix multiplication.

Matrix multiplication is not only quite a common problem, but it is also very useful as an example, as it is easy to understand and reveals most of the issues.

More Matrix Multiplication

$$A_{ij} = \sum_{k=1,N} B_{ik}C_{kj}$$

So to form the product of two $N \times N$ square matrices takes N^3 multiplications and N^3 additions. There are no clever techniques for reducing this computational work significantly (save eliminating about N^2 additions, which is of little consequence).

The amount of memory occupied by the matrices scales as N^2, and is exactly $24N^2$ bytes assuming all are distinct and double precision.

Most of these examples use $N = 2048$, so require around 100MB of memory, and will take 16s if run at 1 GFLOPs.

Our Computer

These examples use a 2.4GHz quad core Core2 with 4GB of RAM. Each core can complete two additions and two multiplications per clock cycle, so its theoretical sustained performance is 9.6 GFLOPs.

Measured memory bandwidth for unit stride access over an array of 64MB is 6GB/s, and for access with a stride of 2048 doubles it is 84MB/s (one item every 95ns).

We will also consider something older and simpler, a 2.8GHz Pentium 4 with 3GB of RAM. Theoretical sustained performance is 5.6 GFLOPs, 4.2GB/s and 104ns. Its data in the following slides will be shown in italics in square brackets.

The Core 2 processor used, a Q6600, was first released in 2007. The Pentium 4 used was first released in 2002. The successor to the Core 2, the Nehalem, was first released late in 2008.

Speeds

```
do i=1,n
  do j=1,n
    t=0
    do k=1,n
      t=t+b(i,k)*c(k,j)
    enddo
    a(i,j)=t
  enddo
enddo
```

```
for(i=0;i<n;i++){
  for(j=0;j<n;j++){
    t=0;
    for(k=0;k<n;k++){
      t+=b[i][k]*c[k][j];
    }
    a[i][j]=t;
  }
}
```

If the inner loop is constrained by the compute power of the processor, it will achieve 9.6 GFLOPs. *[5.6 GFLOPS]*

If constrained by bandwidth, loading two doubles and performing two FLOPS per iteration, it will achieve 750 MFLOPs. *[520 MFLOPS]*

If constrained by the large stride access, it will achieve two FLOPs every 95ns, or 21 MFLOPs. *[19 MFLOPS]*

The First Result

When compiled with `gfortran -O0` the code achieved 41.6 MFLOPS. *[37 MFLOPS]*

The code could barely be less optimal – even `t` was written out to memory, and read in from memory, on each iteration. The processor has done an excellent job with the code to achieve 47ns per iteration of the inner loop. This must be the result of some degree of speculative loading overlapping the expected 95ns latency.

In the mess which follows, one can readily identify the memory location `-40(%rbp)` with `t`, and one can also see two integer multiplies as the offsets of the elements `b(i,k)` and `c(k,j)` are calculated.

Messy

```
.L22:
        movq    -192(%rbp), %rbx
        movl    -20(%rbp), %esi
        movslq  %esi, %rdi
        movl    -28(%rbp), %esi
        movslq  %esi, %r8
        movq    -144(%rbp), %rsi
        imulq   %r8, %rsi
        addq    %rsi, %rdi
        movq    -184(%rbp), %rsi
        leaq    (%rdi,%rsi), %rsi
        movsd   (%rbx,%rsi,8), %xmm1
        movq    -272(%rbp), %rbx
        movl    -28(%rbp), %esi
        movslq  %esi, %rdi
        movl    -24(%rbp), %esi
        movslq  %esi, %r8
        movq    -224(%rbp), %rsi
        imulq   %r8, %rsi
        addq    %rsi, %rdi
        movq    -264(%rbp), %rsi
        leaq    (%rdi,%rsi), %rsi
        movsd   (%rbx,%rsi,8), %xmm0
        mulsd   %xmm1, %xmm0
        movsd   -40(%rbp), %xmm1
        addsd   %xmm1, %xmm0
        movsd   %xmm0, -40(%rbp)
        cmpl    %ecx, -28(%rbp)
        sete    %bl
        movzbl  %bl, %ebx
        addl    $1, -28(%rbp)
        testl   %ebx, %ebx
        je      .L22
```


Faster

When compiled with `gfortran -O1` the code achieved 118 MFLOPS. The much simpler code produced by the compiler has given the processor greater scope for speculation and simultaneous outstanding memory requests. Don't expect older (or more conservative) processors to be this smart – on an ancient Pentium 4 the speed improved from 37.5 MFLOPS to 37.7 MFLOPS.

Notice that `t` is now maintained in a register, `%xmm0`, and not written out to memory on each iteration. The integer multiplications of the previous code have all disappeared, one by conversion into a Shift Arithmetic Left Quadbyte of 11 (i.e. multiply by 2048, or `2^11`).

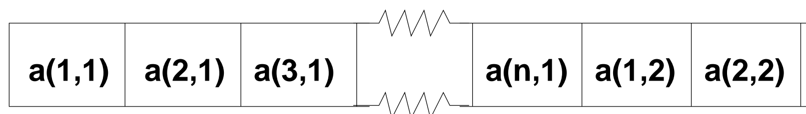
```
.L10:
        movslq  %eax, %rdx
        movq    %rdx, %rcx
        salq    $11, %rcx
        leaq    -2049(%rcx,%r8), %rcx
        addq    %rdi, %rdx
        movsd   0(%rbp,%rcx,8), %xmm1
        mulsd   (%rbx,%rdx,8), %xmm1
        addsd   %xmm1, %xmm0
        addl    $1, %eax
        leal    -1(%rax), %edx
        cmpl    %esi, %edx
        jne     .L10
```

Unrolling: not faster

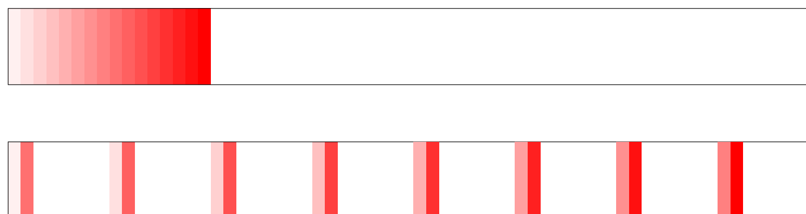
```
do i=1,nn
  do j=1,nn
    t=0
    do k=1,nn,2
      t=t+b(i,k)*c(k,j)+b(i,k+1)*c(k+1,j)
    enddo
    a(i,j)=t
  enddo
enddo
```

This ‘optimisation’ reduces the overhead of testing the loop exit condition, and little else. The memory access pattern is unchanged, and the speed is also pretty much unchanged – up by about 4%.

Memory Access Pattern



Below an 8×8 array being accessed the correct and incorrect way around.



Blocking: Faster

```
do i=1,nn,2
  do j=1,nn
    t1=0
    t2=0
    do k=1,nn
      t1=t1+b(i,k)*c(k,j)
      t2=t2+b(i+1,k)*c(k,j)
    enddo
    a(i,j)=t1
    a(i+1,j)=t2
  enddo
enddo
```

This has changed the memory access pattern on the array `b`. Rather than the pessimal order
`b(1,1) b(1,2) b(1,3) b(1,4) ... b(1,n) b(2,1) b(2,2)`
we now have
`b(1,1) b(2,1) b(1,2) b(2,2) ..... b(1,n) b(2,n) b(3,1) b(4,1)`
Every other item is fetched almost for free, because its immediate neighbour has just been fetched. The number of iterations within this inner loop is the same, but the loop is now executed half as many times.

Yes, Faster

We would predict a speedup of about a factor of two, and that is indeed seen. Now the Core 2 reaches 203 MFLOPS (up from 118 MFLOPS), and the Pentium 4 71 MFLOPS (up from 38 MFLOPS).

Surprisingly changing the blocking factor from 2 to 4 (i.e. four elements calculated in the inner loop) did not impress the Core 2. It improved to just 224 MFLOPS (+10%). The Pentium 4, which had been playing fewer clever tricks in its memory controller, was much happier to see the blocking factor raised to 4, now achieving 113 MFLOPS (+59%).

More, more more!

```
do i=1,nn,nb
  do j=1,nn
    do kk=0,nb-1
      a(i+kk,j)=0
    enddo
    do k=1,nn
      do kk=0,nb-1
        a(i+kk,j)=a(i+kk,j)+b(i+kk,k)*c(k,j)
      enddo
    enddo
  enddo
enddo
```

With `nb=1` this code is mostly equivalent to our original naïve code. Only less readable, potentially buggier, more awkward for the compiler, and `a(i,j)` is now unlikely to be cached in a register. With `nb=1` the Core 2 achieves 74 MFLOPS, and the Pentium 4 33 MFLOPS. But with `nb=64` the Core 2 achieves 530 MFLOPS, and the Pentium 4 320 MFLOPS – their best scores so far.

Better, better, better

```
do k=1,nn,2
  do kk=0,nb-1
    a(i+kk,j)=a(i+kk,j)+b(i+kk,k)*c(k,j)+   &
                        b(i+kk,k+1)*c(k+1,j)
  enddo
enddo
```

Fewer loads and stores on `a(i,j)`, and the Core 2 likes this, getting 707 MFLOPS. The Pentium 4 now manages 421 MFLOPS. Again this is trivially extended to a step of four in the `k` loop, which achieves 750 MFLOPS *[448 MFLOPS]*

Other Orders

```
a=0
do j=1,nn
  do k=1,nn
    do i=1,nn
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

Much better. 1 GFLOPS on the Core 2, and 660 MFLOPS on the Pentium 4.

In the inner loop, `c(k,j)` is constant, and so we have two loads and one store, all unit stride, with one add and one multiply.

Better Yet

```
a=0
do j=1,nn,2
  do k=1,nn
    do i=1,nn
      a(i,j)=a(i,j)+b(i,k)*c(k,j)+ &
                    b(i,k)*c(k,j+1)
    enddo
  enddo
enddo
```

Now the inner loop has `c(k,j)` and `c(k,j+1)` constant, so still has two loads and one store, all unit stride (assuming efficient use of registers), but now has two adds and two multiplies.

Both processors love this – 1.48 GFLOPS on the Core 2, and 1.21 GFLOPS on the Pentium 4.

Limits

Should we exend this by another factor of two, and make the outer loop of step 4?

The Core 2 says a clear yes, improving to 1.93 GFLOPS (+30%). The Pentium 4 is less enthusiastic, improving to 1.36 GFLOPS (+12%).

What about 8? The Core 2 then gives 2.33 GFLOPS (+20%), and the Pentium 4 1.45 GFLOPS (+6.6%).

Spills

With a step of eight in the outer loop, there are eight constants in the inner loop, `c(k,j)` to `c(k,j+7)`, as well as the two variables `a(i,j)` and `b(i,k)`. The Pentium 4 has just run out of registers, so three of the constant `c`'s have to be loaded from memory (cache) as they don't fit into registers.

The Core 2 has twice as many FP registers, so has not suffered what is called a 'register spill', when values which ideally would be kept in registers spill back into memory as the compiler runs out of register space.

Horrid!

Are the above examples correct? Probably not – I did not bother to test them!

The concepts are correct, but the room for error in coding in the above style is large. Also the above examples assume that the matrix size is divisible by the block size. General code needs (nasty) sections for tidying up when this is not the case.

Also, we are achieving around 20% of the peak performance of the processor. Better than the initial 1-2%, but hardly wonderful.

Best Practice

Be lazy. Use someone else's well-tested code where possible.

Using Intel's Maths Kernel Library one achieves 4.67 GFLOPS on the Pentium 4, and 8.88 GFLOPS on one core of a Core 2. Better, that library can make use of multiple cores of the Core 2 with no further effort, then achieving 33.75 GFLOPS when using all four cores.

N.B.

```
call cpu_time(time1)
...
call cpu_time(time2)
write(*,*) time2-time1
```

records total CPU time, so does not show things going faster as more cores are used. One wants wall-clock time:

```
call system_clock(it1,ic)
time1=real(it1,kind(1d0))/ic
...
```

Other Practices

Use Fortran90's `matmul` routine.

Core 2

ifort -O3:	5.10 GFLOPS
gfortran:	3.05 GFLOPS
pathf90 -Ofast:	2.30 GFLOPS
pathf90	1.61 GFLOPS
ifort:	0.65 GFLOPS

Pentium 4

ifort -O3:	1.55 GFLOPS
gfortran:	1.05 GFLOPS
ifort:	0.43 GFLOPS

Lessons

Beating the best professional routines is hard.

Beating the worst isn't.

The variation in performance due to the use of different routines is *much* greater than that due to the single-core performance difference between a Pentium 4 and a Core 2. Indeed, the Pentium 4's best result is about 30$\times$ as fast as the Core 2's worst result.

Difficulties

For the hand-coded tests, the original naïve code on slide 91 compiled with `gfortran -O1` recorded 118 MFLOPS *[37.7 MFLOPS]*, and was firmly beaten by reversing the loop order (slide 98) at 1 GFLOPS *[660 MFLOPS]*.

Suppose we re-run these examples with a matrix size of 25×25 rather than 2048×2048. Now the speeds are 1366 MFLOPS *[974 MFLOPS]* and 1270 MFLOPS *[770 MFLOPS]*.

The three arrays take $3 \times 25 \times 25 \times 8$ bytes, or 15KB, so things fit into L1 cache on both processors. L1 cache is insensitive to data access order, but the 'naïve' method allows a cache access to be converted into a register access (in which a sum is accumulated).

Leave it to Others!

So comparing these two methods, on the Core 2 the one which wins by a factor of 8.5 for the large size is 7% slower for the small size. For the Pentium 4 the results are more extreme: $17\times$ faster for the large case, 20% slower for the small case.

A decent matrix multiplication library will use different methods for different problem sizes, ideally swapping between them at the precisely optimal point. It is also likely that there will be more than two methods used as one moves from very small to very large problems.

Suppose we now try a matrix size of 2×2. The 'naïve' code now manages 400 MFLOPS *[540 MFLOPS]*, and the reversed code 390 MFLOPS *[315 MFLOPS]*.

If instead one writes out all four expressions for the elements of `a` explicitly, the speed jumps to about 3,200 MFLOPS *[1,700 MFLOPS]*.

Loops of unknown (at compile time) but small (at run time) iteration count can be quite costly compared to the same code with the loop entirely eliminated.

For the first test, the 32 bit compiler really did produce significantly better code than the 64 bit compiler, allowing the Pentium 4 to beat the Core 2.

Maintaining Zero GFLOPS

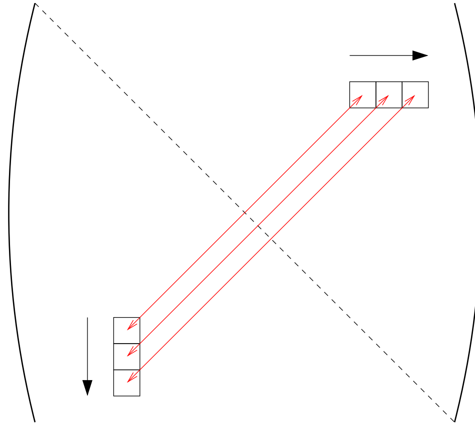
One matrix operation for which one can never exceed zero GFLOPS is the transpose. There are no floating point operations, but the operation still takes time.

```
do i=1,nn
  do j=i+1,nn
    t=a(i,j)
    a(i,j)=a(j,i)
    a(j,i)=t
  enddo
enddo
```

This takes about 24ns per element in `a` on the Core 2 *[96ns on Pentium 4]* with a matrix size of 4096.

Problems

It is easy to see what is causing trouble here. Whereas one of the accesses in the loop is sequential, the other is of stride 32K. We would naïvely predict that this code would take around 43ns *[52ns]* per element, based on one access taking negligible time, and the other the full latency of main memory.

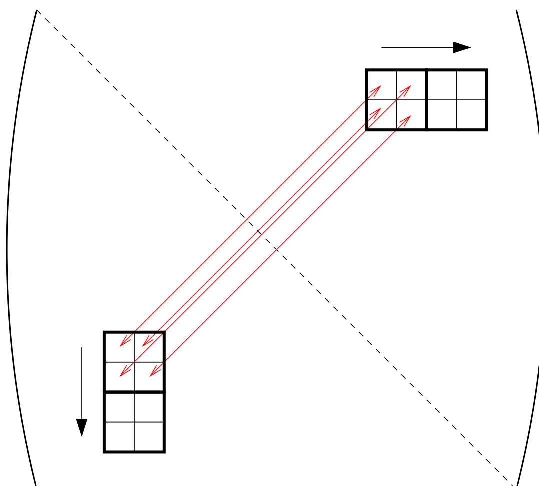


The Pentium 4 is doing worse than our naïve model because 104ns is its access time for *reads* from main memory. Here we have writes as well, so there is a constant need to evict dirty cache lines. This will make things worse.

The Core 2 is showing the sophistication of a memory controller capable of having several outstanding requests and a CPU capable of speculation.

Faster

If the inner loop instead dealt with a small 2×2 block of element, it would have two stride 32K accesses per iteration and exchange eight elements, instead of one stride 32K access to exchange two elements. If the nasty stride is the problem, this should run twice as fast. It does: 12ns per element *[42ns]*.



Nasty Code

```
do i=1,nn,2
  do j=i+2,nn,2
    t=a(i,j)
    a(i,j)=a(j,i)
    a(j,i)=t
    t=a(i+1,j)
    a(i+1,j)=a(j,i+1)
    a(j,i+1)=t
    t=a(i,j+1)
    a(i,j+1)=a(j+1,i)
    a(j+1,i)=t
    t=a(i+1,j+1)
    a(i+1,j+1)=a(j+1,i+1)
    a(j+1,i+1)=t
  enddo
enddo

do i=1,nn,2
  j=i+1
  t=a(i,j)
  a(i,j)=a(j,i)
  a(j,i)=t
enddo
```

Is this even correct? Goodness knows – it is unreadable, and untested. And it is certainly wrong if `nn` is odd.

How far should we go?

Why not use a 3×3 block, or a 10×10 block, or some other $n \times n$ block? For optimum speed one should use a larger block than 2×2.

Ideally we would read in a whole cache line and modify all of it for the sequential part of reading in a block in the lower left of the matrix. Of course, we can't. There is no guarantee that the array starts on a cache line boundary, and certainly no guarantee that each row starts on a cache line boundary.

We also want the whole of the block in the upper right of the matrix to stay in cache whilst we work on it. Not usually a problem – level one cache can hold a couple of thousand doubles, *but* with a matrix size which is a large power of two, `a(i,j)` and `a(i,j+1)` will be separated by a multiple of the cache size, and in a direct mapped cache will be stored in the same cache line.

Different Block Sizes

Block Size	Pentium 4	Athlon II	Core 2
1	100ns	41ns	25ns
2	42ns	22ns	12ns
4	27ns	21ns	11ns
8	22ns	19ns	8ns
16	64ns	17ns	8ns
32	88ns	41ns	9ns
64	102ns	41ns	12ns

Caches:

Pentium 4: L1 16K 4 way, L2 512K 8 way.
Athlon II: L1 64K 2 way, L2 1MB 16 way.
Core 2: L1 32K 8 way, L2 4MB 16 way.

Notice that even on this simple test we have the liberty of saying that the Athlon II is merely 15% faster than the old Pentium 4, or a more respectable $3.75\times$ faster. One can prove almost anything with benchmarks. I have several in which that Athlon II would easily beat that Core 2...

Nastier Code

```
do i=1,nn,nb
  do j=i+nb,nn,nb
    do ii=0,nb-1
      do jj=0,nb-1
        t=a(i+ii,j+jj)
        a(i+ii,j+jj)=a(j+jj,i+ii)
        a(j+jj,i+ii)=t
      enddo
    enddo
  enddo
enddo

do i=1,nn,nb
  j=i
  do ii=0,nb-1
    do jj=ii+1,nb-1
      t=a(i+ii,j+jj)
      a(i+ii,j+jj)=a(j+jj,i+ii)
      a(j+jj,i+ii)=t
    enddo
  enddo
enddo
```

Is this even correct? Goodness knows – it is unreadable, and untested. And it is certainly wrong if `nn` is not divisible by `nb`.

Different Approaches

One can also transpose a square matrix by recursion: divide the matrix into four smaller square submatrices, transpose the two on the diagonal, and transpose and exchange the two off-diagonal submatrices.

For computers which like predictable strides, but don't much care what those strides are (i.e. old vector computers, and maybe GPUs?), one might consider a transpose moving down each off-diagonal in turn, exchanging with the corresponding off-diagonal.

By far the best method is not to transpose at all – make sure that whatever one was going to do next can cope with its input arriving lacking a final transpose.

Note that most routines in the ubiquitous linear algebra package BLAS accept their input matrices in either conventional or transposed form.

There is More Than Multiplication

This lecture has concentrated on the 'trivial' examples of matrix multiplication amd transposes. The idea that different methods need to be used for different problem sizes is much more general, and applies to matrix transposing, solving systems of linear equations, FFTs, etc.

It can make for large, buggy, libraries. For matrix multiplication, the task is valid for multiplying an $n \times m$ matrix by a $m \times p$ matrix. One would hope that any released routine was both correct and fairly optimal for all square matrices, and the common case of one matrix being a vector. However, did the programmer think of testing for the case of multiplying a $1,000,001 \times 3$ matrix by a 3×5 matrix? Probably not. One would hope any released routine was still correct. One might be disappointed by its optimality.

Doing It Oneself

If you are tempted by DIY, it is probably because you are working with a range of problem sizes which is small, and unusual. (Range small, problem probably not small.)

To see if it is worth it, try to estimate the MFLOPS achieved by whatever routine you have readily to hand, and compare it to the processor's peak theoretical performance. This will give you an upper bound on how much faster your code could possibly go. Some processors are notoriously hard to get close to this limit. Note that here the best result for the Core 2 was about 91%, whereas for the Pentium 4 it was only 83%.

If still determined, proceed with a theory text book in one hand, and a stop-watch in the other. And then test the resulting code thoroughly.

Although theory may guide you towards fast algorithms, processors are sufficiently complex and undocumented that the final arbitrator of speed has to be the stopwatch.

Memory Management

Memory: a Programmer's Perspective

From a programmer's perspective memory is simply a linear array into which bytes are stored. The array is indexed by a pointer which runs from 0 to 2^{32} (4GB) on 32 bit machines, or 2^{64} (16EB) on 64 bit machines.

The memory has no idea what type of data it stores: integer, floating point, program code, text, it's all just bytes.

An address may have one of several attributes:

Invalid	not allocated
Read only	for constants and program code
Executable	for program code, not data
Shared	for inter-process communication
On disk	paged to disk to free up real RAM

(Valid virtual addresses on current 64 bit machines reach only 2^{48} (256TB). So far no-one is complaining. To go further would complicate the page table (see below).)

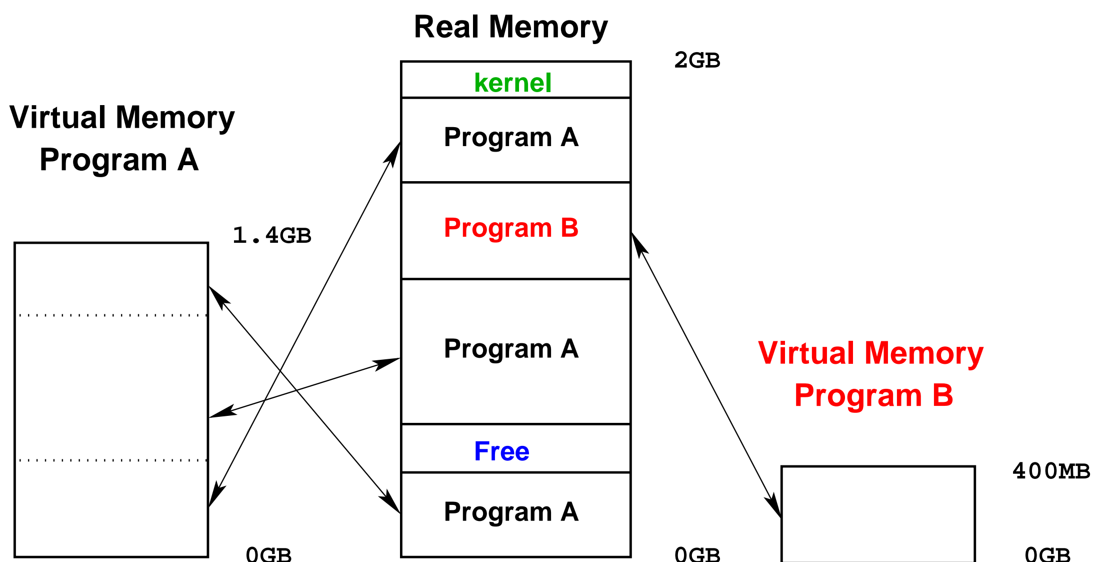
Pages

In practice memory is broken into *pages*, contiguous regions, often of 4KB, which are described by just a single set of the above attributes. When the operating system allocates memory to a program, the allocation must be an integer number of pages. If this results in some extra space, `malloc()` or `allocate()` will notice, and may use that space in a future allocation without troubling the operating system.

Modern programs, especially those written in C or, worse, C++, do a lot of allocating and deallocating of small amounts of memory. Some remarkably efficient procedures have been developed for dealing with this. Ancient programs, such as those written in Fortran 77, do no run-time allocation of memory. All memory is fixed when the program starts.

Pages also allow for a mapping to exist between *virtual* addresses as seen by a process, and *physical* addresses in hardware.

No Fragmentation



Pages also have an associated location in real, physical memory. In this example, program A believes that it has an address space extending from 0MB to 1400MB, and program B believes it has a distinct space extending from 0MB to 400MB. Neither is aware of the mapping of its own virtual address space into physical memory, or whether that mapping is contiguous.

Splendid Isolation

This scheme gives many levels of isolation.

Each process is able to have a contiguous address space, starting at zero, regardless of what other processes are doing.

No process can accidentally access another process's memory, for no process is able to use physical addresses. They have to use virtual addresses, and the operating system will not allow two virtual addresses to map to the same physical address (except when this is really wanted).

If a process attempts to access a virtual address which it has not been granted by the operating system, no mapping to a physical address will exist, and the access must fail. A segmentation fault.

A virtual address is unique only when combined with a process ID (deliberate sharing excepted).

Fast, and Slow

This scheme might appear to be very slow. Every memory access involves a translation from a virtual address to a physical address. Large translation tables (page tables) are stored in memory to assist. These are stored at known locations in physical memory, and the kernel, unlike user processes, can access physical memory directly to avoid a nasty catch-22.

Every CPU has a cache dedicated to storing the results of recently-used page table look-ups, called the TLB. This eliminates most of the speed penalty, except for random memory access patterns.

A TLB is so essential for performance with virtual addressing that the 80386, the first Intel processor to support virtual addressing, had a small (32 entry) TLB, but no other cache.

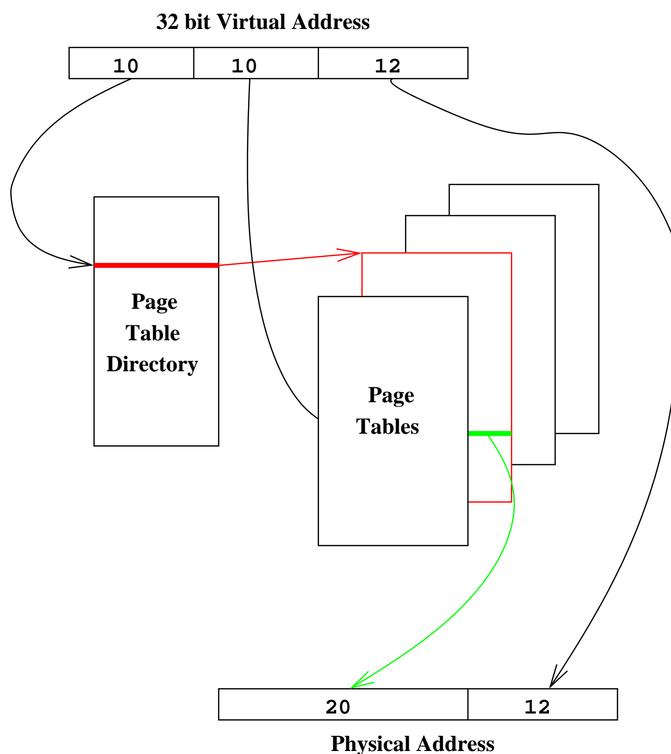
Page Tables

A 32 bit machine will need a four-byte entry in a page table per page. With 4KB pages, this could be done with a 4MB page table per process covering the whole of its virtual address space. However, for processes which make modest use of virtual address space, this would be rather inefficient. It would also be horrific in a 64 (or even 48) bit world.

So the page table is split into two. The top level describes blocks of 1024 pages (4MB). If no address in that range is valid, the top level table simply records this invalidity. If any address is valid, the top level table then points to a second level page table which contains the 1024 entries for that 4MB region. Some of those entries may be invalid, and some valid.

The logic is simple. For a 32 bit address, the top ten bits index the top level page table, the next ten index the second level page table, and the final 12 an address within the 4KB page pointed to by the second level page table.

Page Tables in Action



For a 64 bit machine, page table entries must be eight bytes. So a 4KB page contains just 512 (2^9) entries. Intel currently uses a four level page table for '64 bit' addressing, giving $4 \times 9 + 12 = 48$ bits. The Alpha processor used a three level table and an 8KB page size, giving $3 \times 10 + 13 = 43$ bits.

Efficiency

This is still quite a disaster. Every memory reference now requires two or three additional accesses to perform the virtual to physical address translation.

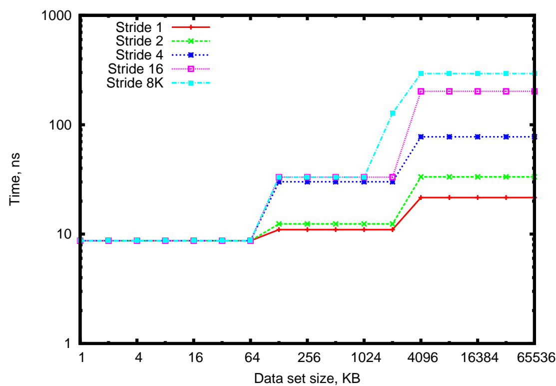
Fortunately, the CPU understands pages sufficiently well that it remembers where to find frequently-referenced pages using a special cache called a TLB. This means that it does not have to keep asking the operating system where a page has been placed.

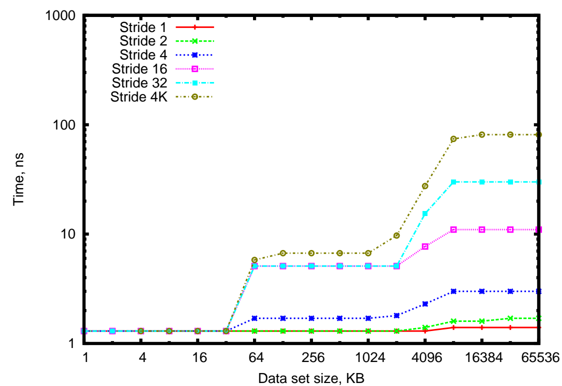
Just like any other cache, TLBs vary in size and associativity, and separate instruction and data TLBs may be used. A TLB rarely contains more than 1024 entries, often far fewer.

Even when a TLB miss occurs, it is rarely necessary to fetch a page table from main memory, as the relevant tables are usually still in secondary cache, left there by a previous miss.

TLB = translation lookaside buffer
ITLB = instruction TLB, DTLB = data TLB if these are separate

TLBs at work





The left is a repeat of the graph on page 80, but with an 8KB stride added. The XP900 uses 8KB pages, and has a 128 entry DTLB. Once the data set is over 1MB, the TLB is too small to hold its pages, and, with an 8KB stride, a TLB miss occurs on every access, adding 92ns.

The right is a repeat of the Core 2 graph from page 81, with a 4KB stride added. The Core 2 uses 4KB pages, and has a 256 entry DTLB. Some more complex interactions are occuring here, but it finishes up with a 50ns penalty.

Given that three levels of page table must be accessed, it is clear that most of the relevant parts of the page table were in cache. So the 92ns and 50ns recovery times for a TLB miss are best cases – with larger data sets it can get worse. The Alpha is losing merely 43 clock cycles, the Core 2 about 120. As the data set gets yet larger, TLB misses will be to page tables not in cache, and random access to a 2GB array results in a memory latency of over 150ns on the Core 2.

More paging

Having suffering one level of translation from virtual to physical addresses, it is conceptually easy to extend the scheme slightly further. Suppose that the OS, when asked to find a page, can go away, read it in from disk to physical memory, and then tell the CPU where it has put it. This is what all modern OSes do (UNIX, OS/2, Win9x / NT, MacOS), and it merely involves putting a little extra information in the page table entry for that page.

If a piece of real memory has not been accessed recently, and memory is in demand, that piece will be paged out to disk, and reclaimed automatically (if slowly) if it is needed again. Such a reclaiming is also called a page fault, although in this case it is not fatal to the program.

Rescuing a page from disk will take about 10ms, compared with under 100ns for hitting main memory. If just one in 10^5 memory accesses involve a page-in, the code will run at half speed, and the disk will be audibly 'thrashing'.

The union of physical memory and the page area on disk is called *virtual memory*. Virtual addressing is a prerequisite for virtual memory, but the terms are not identical.

Less paging

Certain pages should not be paged to disk. The page tables themselves are an obvious example, as is much of the kernel and parts of the disk cache.

Most OSes (including UNIX) have a concept of a *locked*, that is, unpageable, page. Clearly all the locked pages must fit into physical memory, so they are considered to be a scarce resource. On UNIX only the kernel or a process running with root privilege can cause its pages to be locked.

Much I/O requires locked pages too. If a network card or disk drive wishes to write some data into memory, it is too dumb to care about virtual addressing, and will write straight to a physical address. With locked pages such pages are easily reserved.

Certain 'real time' programs which do not want the long delays associated with recovering pages from disk request that their pages are locked. Examples include CD/DVD writing software, or video players.

Blatant Lies

Paging to disk as above enables a computer to pretend that it has more RAM than it really does. This trick can be taken one stage further. Many OSes are quite happy to allocate virtual address space, leaving a page table entry which says that the address is valid, not yet ever been used, and has no physical storage associated with it. Physical storage will be allocated on first use. This means that a program will happily pass all its `malloc()` / `allocate` statements, and only run into trouble when it starts trying to use the memory.

The `ps` command reports both the virtual and physical memory used:

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
spqr1 20241  100 12.7 711764 515656 pts/9  Rl+  13:36   3:47 castep si64
```

RSS – Resident Set Size (i.e. physical memory use). Will be less than the physical memory in the machine. `%MEM` is the ratio of this to the physical memory of the machine, and thus can never exceed 100.

VSZ – Virtual SiZe, i.e. total virtual address space allocated. Cannot be smaller than RSS.

The Problem with Lying

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS      TTY   STAT START   TIME COMMAND
spqr1 25175 98.7 25.9 4207744 1049228 pts/3 R+   14:02  0:15 ./a.out
```

Currently this is fine – the process is using just under 26% of the memory. However, the `VSZ` field suggests that it has been promised 104% of the physical memory. This could be awkward.

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS      TTY   STAT START   TIME COMMAND
spqr1 25175 39.0 90.3 4207744 3658252 pts/0 D+   14:02  0:25 ./a.out
```

Awkward. Although the process does no I/O its status is 'D' (waiting for 'disk'), its share of CPU time has dropped (though no other process is active), and inactive processes have been badly squeezed. At this point Firefox had an RSS of under 2MB and was extremely slow to respond. It had over 50MB before it was squeezed.

Interactive users will now be very unhappy, and if the computer had another GB that program would run almost three times faster.

Grey Areas – How Big is Too Big?

It is hard to say precisely. If a program allocates one huge array, and then jumps randomly all over it, then the entirety of that array must fit into physical memory, or there will be a huge penalty. If a program allocates two large arrays, spends several hours with the first, then moves it attention to the second, the penalty if only one fits into physical memory at a time is slight. Total usage of physical memory is reported by `free` under Linux. Precise interpretation of the fields is still hard.

```
$ free
             total       used       free     shared    buffers     cached
Mem:       4050700     411744    3638956          0       8348     142724
-/+ buffers/cache:     260672    3790028
Swap:      6072564      52980    6019584
```

The above is fine. The below isn't. Don't wait for `free` to hit zero – it won't.

```
$ free
             total       used       free     shared    buffers     cached
Mem:       4050700    4021984      28716          0        184     145536
-/+ buffers/cache:    3876264     174436
Swap:      6072564     509192    5563372
```


Page sizes

A page is the smallest unit of memory allocation from OS to process, and the smallest unit which can be paged to disk. Large page sizes result in wasted memory from allocations being rounded up, longer disk page in and out times, and a coarser granularity on which unused areas of memory can be detected and paged out to disk. Small page sizes lead to more TLB misses, as the virtual address space 'covered' by the TLB is the number of TLB entries multiplied by the page size.

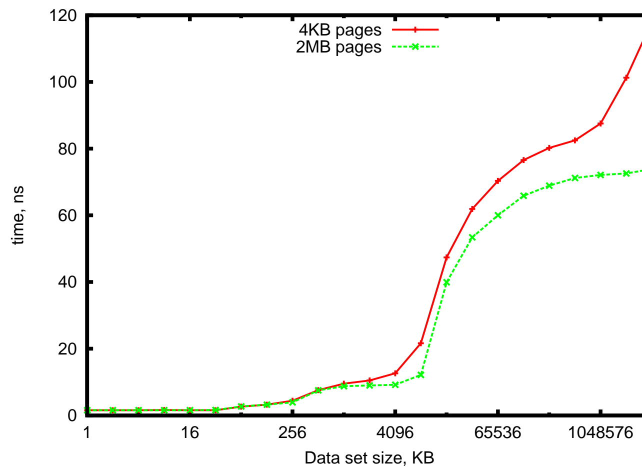
Large-scale scientific codes which allocate hundreds of MB of memory benefit from much larger page sizes than a mere 4KB. However, a typical UNIX system has several dozen small processes running on it which would not benefit from a page size of a few MB.

Intel's processors do support 2MB pages, but support in Linux is unimpressive prior to 2.6.38. Support from Solaris for the page sizes offered by the (ancient) UltraSPARC III (8K, 64K, 512K and 4MB) is much better.

DEC's Alpha solves this issue in another fashion, by allowing one TLB entry to refer to one, eight, 64 or 512 consecutive pages, thus effectively increasing the page size.

Large Pages in Linux

From kernel 2.6.38, Linux will use large pages (2MB) by default when it can. This reduces TLB misses when jumping randomly over large arrays.



The disadvantage is that sometimes fragmentation in physical memory will prevent Linux from using (as many) large pages. This will make code run slower, and the poor programmer will have no idea what has happened.

This graph can be compared with that on page 129, noting that here a random access pattern is used, the y axis is not logarithmic, the processor is an Intel Sandy Bridge, and the x axis is extended another factor of 64.

Expectations

The Sandy Bridge CPU used to generate that graph has a 32KB L1 cache, a 256KB L2, and a 8MB L3. If one assumes that the access times are 1.55ns, 3.9ns, 9.5ns for those, and for main memory 72.5ns, then the line for 2MB pages can be reproduced remarkably accurately. (E.g. at 32MB assume one quarter of accesses are lucky and are cached in L3 (9.5ns), the rest are main memory (72.5ns), so expect 56.7ns. Measured 53.4ns.)

With 4KB pages, the latency starts to increase again beyond about 512MB. The cause is the last level of the page table being increasingly likely to have been evicted from the last level of cache by the random access on the data array. If the TLB miss requires a reference to a part of the page table in main memory, it must take at least 72ns. This is probably happening about half of the time for the final data point (4GB).

This graph shows very clearly that ‘toy’ computers hate big problems: accessing large datasets can be *much* slower than accessing smaller ones, although the future is looking (slightly) brighter.

Caches and Virtual Addresses

Suppose we have a two-way associative 2MB cache. This means that we can cache any contiguous 2MB region of physical memory, and any two physical addresses which are identical in their last 20 bits.

Programs works on virtual addresses. The mapping from virtual to physical preserves the last 12 bits (assuming 4KB pages), but is otherwise unpredictable. A 2MB region of virtual address space will be completely cacheable only for some mappings. If one is really unlucky, a mere 12KB region of virtual address space will map to three physical pages whose last 20 bits are all identical. Then this cannot be cached. A random virtual to physical mapping would make caching all of a 2MB region very unlikely.

Most OSes do magic (page colouring) which reduces, or eliminates, this problem, but Linux does not. This is particularly important if a CPU's L1 cache is larger than its associativity multiplied by the OS's page size (AMD Athlon / Opteron, but not Intel). When the problem is not eliminated, one sees variations in runtimes as a program is run repeatedly (and the virtual to physical mapping changes), and the expected sharp steps in performance as arrays grow larger than caches are slurred.

Segments

A program uses memory for many different things. For instance:

- The code itself
- Shared libraries
- Statically allocated uninitialised data
- Statically allocated initialised data
- Dynamically allocated data
- Temporary storage of arguments to function calls and of local variables

These areas have different requirements.

Segments

Text

Executable program code, including code from statically-linked libraries. Sometimes constant data ends up here, for this segment is read-only.

Data

Initialised data (numeric and string), from program and statically-linked libraries.

BSS

Uninitialised data of fixed size. Unlike the data segment, this will not form part of the executable file. Unlike the heap, the segment is of fixed size.

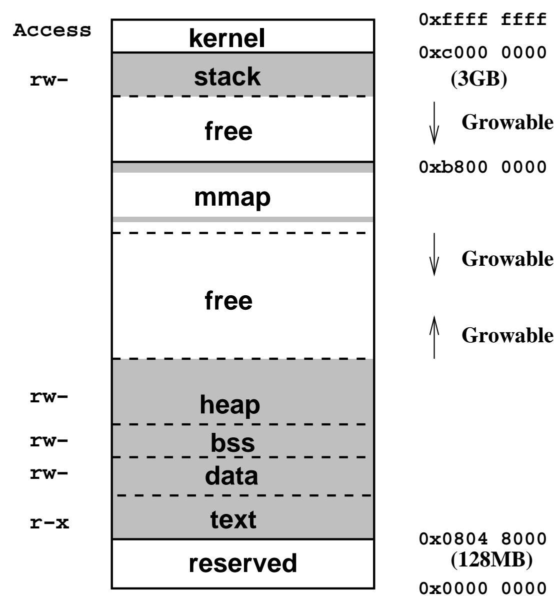
heap

Area from which `malloc()` / `allocate()` traditionally gain memory.

stack

Area for local temporary variables in (recursive) functions, function return addresses, and arguments passed to functions.

A Linux Memory Map



This is roughly the layout used by Linux 2.6 on 32 bit machines, and *not to scale*.

The `mmap` region deals with shared libraries and large objects allocated via `malloc`, whereas smaller `malloc`ed objects are placed on the heap in the usual fashion. Earlier versions grew the `mmap` region *upwards* from about 1GB (0x4000 0000).

Note the area around zero is reserved. This is so that null pointer dereferencing will fail: ask a C programmer why this is important.

What Went Where?

Determining to which of the above data segments a piece of data has been assigned can be difficult. One would strongly expect C's `malloc` and F90's `allocate` to reserve space on the heap. Likewise small local variables tend to end up on the stack.

Large local variables really ought not go on the stack: it is optimised for the low-overhead allocation and deletion needed for dealing with lots of small things, but performs badly when a large object lands on it. However compilers sometimes get it wrong.

UNIX limits the size of the stack segment and the heap, which it 'helpfully' calls 'data' at this point. See the '`ulimit`' command (`[ba]sh`).

Because `ulimit` is an internal shell command, it is documented in the shell man pages (e.g. '`man bash`'), and does not have its own man page.

Sharing

If multiple copies of the same program or library are required in memory, it would be wasteful to store multiple identical copies of their unmodifiable read-only pages. Hence many OSes, including UNIX, keep just one copy in memory, and have many virtual addresses refering to the same physical address. A count is kept, to avoid freeing the physical memory until no process is using it any more!

UNIX does this for shared libraries and for executables. Thus the memory required to run three copies of Firefox is less than three times the memory required to run one, even if the three are being run by different users.

Two programs are considered identical by UNIX if they are on the same device and have the same inode. See elsewhere for a definition of an inode.

If an area of memory is shared, the `ps` command apportions it appropriately when reporting the RSS size. If the whole libc is being shared by ten processes, each gets merely 10% accounted to it.

mmap

It has been shown that the OS can move data from physical memory to disk, and transparently move it back as needed. However, there is also an interface for doing this explicitly. The `mmap` system call requests that the kernel set up some page tables so that a region of virtual address space is mapped onto a particular file. Thereafter reads and writes to that area of 'memory' actually go through to the underlying file.

The reason this is of interest, even to Fortran programmers, is that it is how all executable files and shared libraries are loaded. It is also how large dynamic objects, such as the result of large `allocate` / `malloc` calls, get allocated. They get a special form of `mmap` which has no physical file associated with it.

Heap vs mmap

Consider the following code:

```
a=malloc(1024*1024*1024); b=malloc(1); free(a)
```

(in the real world one assumes that something else would occur before the final `free`).

With a single heap, the heap now has 1GB of free space, followed by a single byte which is in use. Because the heap is a single contiguous object with just one moveable end, there is no way of telling the OS that is can reclaim the unused 1GB. That memory will remain with the program and be available for its future allocations. The OS does not know that its current contents are no longer required, so its contents must be preserved, either in physical memory or in a page file. If the program (erroneously) tries accessing that freed area, it will succeed.

Had the larger request resulted in a separate object via `mmap`, then the `free` would have told the kernel to discard the memory, and to ensure that any future erroneous accesses to it result in segfaults.

Automatically done

Currently by default objects larger than 128KB allocated via `malloc` are allocated using `mmap`, rather than via the heap. The size of allocation resulting will be rounded up to the next multiple of the page size (4KB). Most Fortran runtime libraries end up calling `malloc` in response to `allocate`. A few do their own heap management, and only call `brk`, which is the basic call to change the size of the heap with no concept of separate objects existing within the heap.

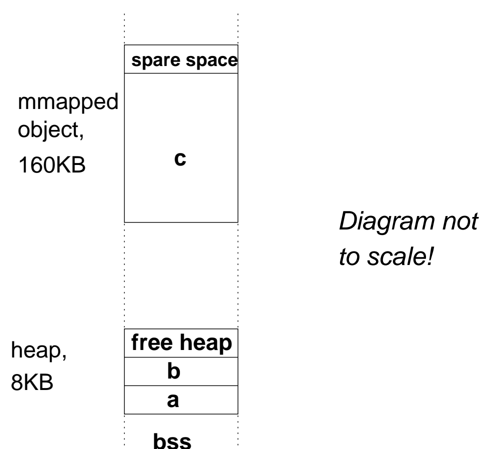
Fortran 90 has an unpleasant habit of placing large temporary and local objects on the stack. This can cause problems, and can be tuned with options such as `-heap-arrays` (ifort) and `-static-data` (Open64).

Objects allocated via `mmap` get placed in a region which lies between the heap and the stack. On 32 bit machines this can lead to the heap (or stack) colliding with this region.

Heap layout

```
double precision, allocatable :: a(:),b(:),c(:)
allocate (a(300),b(300),c(20000))
```

In the absence of other allocations, one would expect the heap to contain `a` followed by `b`. This is 600 doubles, 4,800 bytes, so the heap will be rounded to 8KB (1024 doubles), the next multiple of 4KB. The array `c`, being over 128KB, will go into a separate object via `mmap`, and this will be 160KB, holding 20,480 doubles.



More segfaults

So attempts to access elements of `c` between one and 20,480 will work, and for `a` indices between one and 300 will find `a`, between 301 and 600 will find `b`, and 601 and 1024 will find free space. Only `a(1025)` will cause a segfault. For indices less than one, `c(0)` would be expected to fail, but `b(-100)` would succeed, and probably hit `a(200)`. And `a(-100)` is probably somewhere in the static data section preceeding the heap, and fine.

Array overwriting can go on for a long while before segfaults occur, unless a pointer gets overwritten, and then dereferenced, in which case the resulting address is usually invalid, particularly in a 64 bit world where the proportion of 64 bit numbers which are valid addresses is low.

Fortran compilers almost always support a `-C` option for checking array bounds. It very significantly slows down array accesses – use it for debugging, not real work! The `-g` option increases the chance that line numbers get reported, but compilers differ in how much information does get reported.

C programmers using `malloc()` are harder to help. But they may wish to ask Google about Electric Fence.

Theory in Practice

```
$ cat test.f90
double precision, allocatable :: a(:),b(:),c(:)

allocate (a(300),b(300),c(20000))
a=0
b(-100)=5

write(*,*)'Maximum value in a is ',maxval(a), &
         ' at location ',maxloc(a)
end

$ ifort test.f90 ; ./a.out
 Maximum value in a is  5.00000000000000  at location  202
$ f95 test.f90 ; ./a.out
 Maximum value in a is  5.0  at location  204
$ gfortran test.f90 ; ./a.out
 Maximum value in a is  5.0000000000000000  at location  202
$ openf90 test.f90 ; ./a.out
 Maximum value in a is  0.E+0  at location  1
```

```
$ ifort -C -g test.f90 ; ./a.out
forrtl: severe (408): fort: (3): Subscript #1 of the array B
has value -100 which is less than the lower bound of 1

$ f95 -C -g test.f90 ; ./a.out
 ******  FORTRAN RUN-TIME SYSTEM  ******
Subscript out of range. Location:  line 5 column 3 of 'test.f90'
Subscript number 1 has value -100 in array 'B'
Aborted

$ gfortran -C -g test.f90 ; ./a.out
 Maximum value in a is  5.0000000000000000   at location   202
$ gfortran -fcheck=bounds -g test.f90 ; ./a.out
At line 5 of file test.f90
Fortran runtime error: Index '-100' of dimension 1 of array 'b'
below lower bound of 1

$ openf90 -C -g test.f90 ; ./a.out
lib-4964 : WARNING
  Subscript is out of range for dimension 1 for array
  'B' at line 5 in file 'test.f90',
  diagnosed in routine '__f90_bounds_check'.
 Maximum value in a is  0.E+0  at location  1
```


Disclaimer

By the time you see this, it is unlikely that any of the above examples is with the current version of the compiler used. These examples are intended to demonstrate that different compilers are different. That is why I have quite a collection of them!

`ifort`: Intel's compiler, v 11.1
`f95`: Sun's compiler, Solaris Studio 12.2
`gfortran`: Gnu's compiler, v 4.5
`openf90`: Open64 compiler, v 4.2.4

Four compilers. Only two managed to report line number, and which array bound was exceeded, and the value of the errant index.

The Stack Layout

Address	Contents	Frame Owner
	...	calling
	2nd argument	function
`%ebp+8`	1st argument	
`%ebp+4`	return address	
`%ebp`	previous `%ebp`	
	local	current
	variables	function
	etc.	
`%esp`	end of stack	

The stack grows downwards, and is divided into frames, each frame belonging to a function which is part of the current call tree. Two registers are devoted to keeping it in order.

Memory Maps in Action

Under Linux, one simply needs to examine `/proc/[pid]/maps` using `less` to see a snapshot of the memory map for any process one owns. It also clearly lists shared libraries in use, and some of the open files. Unfortunately it lists things upside-down compared to our pictures above.

The example on the next page clearly shows a program with the bottom four segments being text, data, bss and heap, of which text and bss are read-only. In this case mmaped objects are growing downwards from `f776 c000`, starting with shared libraries, and then including large malloced objects.

The example was from a 32 bit program running on 64 bit hardware and OS. In this case the kernel does not need to reserve such a large amount of space for itself, hence the stack is able to start at `0xfffb 9000` not `0xc000 0000`, and the start of the `mmap` region also moves up by almost 1GB.

Files in `/proc` are not real files, in that they are not physically present on any disk drive. Rather attempts to read from these 'files' are interpretted by the OS as requests for information about processes or other aspects of the system.

The machine used here does not set read and execute attributes separately – any readable page is executable.

The Small Print

```
$ tac /proc/20777/maps
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
fff6e000-fffb9000 rwxp 00000000 00:00 0          [stack]
f776b000-f776c000 rwxp 0001f000 08:01 435109     /lib/ld-2.11.2.so
f776a000-f776b000 r-xp 0001e000 08:01 435109     /lib/ld-2.11.2.so
f7769000-f776a000 rwxp 00000000 00:00 0
f774b000-f7769000 r-xp 00000000 08:01 435109     /lib/ld-2.11.2.so
f7744000-f774b000 rwxp 00000000 00:00 0
f773e000-f7744000 rwxp 00075000 00:13 26596314   /opt/intel/11.1-059/lib/ia32/libguide.so
f76c8000-f773e000 r-xp 00000000 00:13 26596314   /opt/intel/11.1-059/lib/ia32/libguide.so
f76a7000-f76a9000 rwxp 00000000 00:00 0
f76a6000-f76a7000 rwxp 00017000 08:01 435034     /lib/libpthread-2.11.2.so
f76a5000-f76a6000 r-xp 00016000 08:01 435034     /lib/libpthread-2.11.2.so
f768e000-f76a5000 r-xp 00000000 08:01 435034     /lib/libpthread-2.11.2.so
f768d000-f768e000 rwxp 00028000 08:01 435136     /lib/libm-2.11.2.so
f768c000-f768d000 r-xp 00027000 08:01 435136     /lib/libm-2.11.2.so
f7664000-f768c000 r-xp 00000000 08:01 435136     /lib/libm-2.11.2.so
f7661000-f7664000 rwxp 00000000 00:00 0
f7660000-f7661000 rwxp 00166000 08:01 435035     /lib/libc-2.11.2.so
f765e000-f7660000 r-xp 00164000 08:01 435035     /lib/libc-2.11.2.so
f765d000-f765e000 ---p 00164000 08:01 435035     /lib/libc-2.11.2.so
f74f9000-f765d000 r-xp 00000000 08:01 435035     /lib/libc-2.11.2.so
f74d4000-f74d5000 rwxp 00000000 00:00 0
f6fac000-f728a000 rwxp 00000000 00:00 0
f6cec000-f6df4000 rwxp 00000000 00:00 0
f6c6b000-f6c7b000 rwxp 00000000 00:00 0
f6c6a000-f6c6b000 ---p 00000000 00:00 0
f6913000-f6b13000 rwxp 00000000 00:00 0
f6912000-f6913000 ---p 00000000 00:00 0
f6775000-f6912000 rwxp 00000000 00:00 0
097ea000-0ab03000 rwxp 00000000 00:00 0          [heap]
0975c000-097ea000 rwxp 01713000 08:06 9319119    /scratch/castep
0975b000-0975c000 r-xp 01712000 08:06 9319119    /scratch/castep
08048000-0975b000 r-xp 00000000 08:06 9319119    /scratch/castep
```


The Madness of C

```
#include<stdio.h>
#include<stdlib.h>

void foo(int *a, int *b);

int main(void){
  int *a,*b;

  a=malloc(sizeof(int));
  b=malloc(sizeof(int));

  *a=2;*b=3;

  printf("The function main starts at address  %.8p\n",main);
  printf("The function foo  starts at address  %.8p\n",foo);

  printf("Before call:\n\n");
  printf("a is a pointer. It is stored at address %.8p\n",&a);
  printf("                It points to address    %.8p\n",a);
  printf("                It points to the value  %d\n",*a);
  printf("b is a pointer. It is stored at address %.8p\n",&b);
  printf("                It points to address    %.8p\n",b);
  printf("                It points to the value  %d\n",*b);
```

```
  foo(a,b);

  printf("\nAfter call:\n\n");
  printf("                 a points to the value  %d\n",*a);
  printf("                 b points to the value  %d\n",*b);

  return 0;
}

void foo(int *c, int *d){

  printf("\nIn function:\n\n");

  printf("Our return address is                    %.8p\n\n",*(&c-1));

  printf("c is a pointer. It is stored at address %.8p\n",&c);
  printf("                It points to address    %.8p\n",c);
  printf("                It points to the value  %d\n",*c);
  printf("d is a pointer. It is stored at address %.8p\n",&d);
  printf("                It points to address    %.8p\n",d);
  printf("                It points to the value  %d\n",*d);

  *c=5;
  *(*(&c+1))=6;
}
```


The Results of Madness

```
The function main starts at address  0x08048484
The function foo  starts at address  0x080485ce
Before call:

a is a pointer. It is stored at address 0xbfdf8dac
                It points to address    0x0804b008
                It points to the value  2
b is a pointer. It is stored at address 0xbfdf8da8
                It points to address    0x0804b018
                It points to the value  3

In function:

Our return address is                   0x0804858d

c is a pointer. It is stored at address 0xbfdf8d90
                It points to address    0x0804b008
                It points to the value  2
d is a pointer. It is stored at address 0xbfdf8d94
                It points to address    0x0804b018
                It points to the value  3

After call:

                 a points to the value  5
                 b points to the value  6
```

The Explanation

```
0xbfdf ffff   approximate start of stack
   ....
0xbfbf 8da8   local variables in main()
   ....
0xbfdf 8d94   second argument to function foo()
0xbfdf 8d90   first argument
0xbfdf 8d8c   return address
   ....
0x0fdf 8d??   end of stack

0x0804 b020   end of heap
0x0804 b018   the value of b is stored here
0x0804 b008   the value of a is stored here
0x0804 b000   start of heap

0x0804 85ce   start of foo() in text segment
0x0804 858d   point at which main() calls foo()
0x0804 8484   start of main() in text segment
```

And if you note nothing else, note that the function foo managed to manipulate its second argument using merely its first argument.

(This example assumes a 32-bit world for simplicity.)

Hello: My First Program

Hello, World

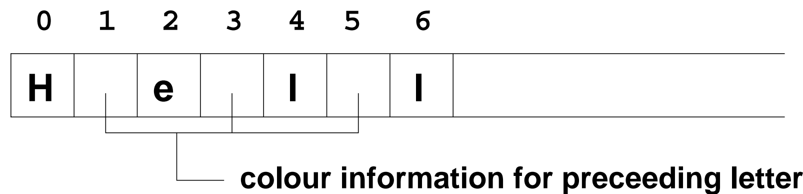
The idea of a first example program being one to print the text “hello, world” is mainly due to the first example in Kernighan and Ritchie’s book “The C Programming Language.” (Kernighan and Ritchie invented C, with most of the work being done by 1973. Before C was standardised by a proper standards’ body (ANSI, in 1989), their book (published 1978) was the definitive description of the language.)

This section considers many ways of writing such a program, and, so that it is as clear as possible what is really happening, most of the examples are in assembler. The first does not even make use of the operating system to do more than act as a program loader.

!!WARNING!! Some of the examples in this section work only on very specific OS versions, although the concepts are much more general. All the Linux examples assume 32 bit Intel Linux, not 64 bit.

Direct Hardware Access

On an IBM PC, the default text video mode is 80 columns by 25 lines. The video memory is mapped starting at address 0xB8000 (top left of screen), with alternate bytes being the ASCII(-ish) representation of the character, and an attribute byte which specifies the colour.



As the screen usually scrolls by one line immediately after a command finishes, we shall print the string on the second line, so starting 160 bytes into the video memory.

DOS Memory

DOS runs on 16 bit computers, so addresses are 16 bits in size, and only 64KB of memory can be addressed. As this was ridiculous, even in the early 1980s, it uses a trick of combining two registers, a segment register and an offset register, in order to construct an address. The physical address is given by $16\times$ segment value + offset, allowing addresses of up to 1MB (20 bits).

One would say that this is an historical irrelevance if it were not for the fact that all Intel's 32 bit processors from the Pentium Pro onwards use something called PAE (Physical Address Extension) to allow them to break the 4GB barrier of 32 bit addressing and address up to 64GB of memory. This uses a similar sort of segment plus offset trick.

PAE is supported by the 32 bit versions of Linux, MacOS X and some versions of Windows Server 2000 and 2003.

DOS .COM files

The simplest executable file format is the DOS `.COM` file. Its contents are simply loaded into a segment at an offset of 0x100 within that segment, all segment registers are pointed to that segment, and the instruction pointer is set to address 0x100 to commence execution.

So a `.COM` file contains no header information, must fit within a 64KB segment, and is always loaded at the same address.

Hello, World (1)

```
section .text
org 0x100                void main(){
                            int cx;
  mov ax,0xB800             char *di, *si;
  mov es,ax
  mov di,160                di=(char*)(0xb8000+160);

  lea si,[string]           si="Hello, World";
  mov cx,12                 cx=12;

next_ch:  movsb             do { *(di++)=*(si++);
  inc di                       di++;
  loop next_ch                 cx--;} while (cx>0);

  ret                    }

string db "Hello, World"
```

Hello, PC World (1)

Tell the assember that this is the text segment, and it starts at 0x100.

The address we wish to write the string to is 0xB800:160 – set this up in `es:di`.

Point `si` at the start of our string, and put the number of characters in `cx`.

The `movsb` instruction is a horrible CISCy thing. It reads a byte from `ds:si`, writes it to `es:di`, and adds one to both `si` and `di`.

We need to skip the attribute bytes in the video memory, so `di` is incremented again.

Finally `loop` is another CISCy thing. It decrements `cx`, and jumps to the label given if `cx` is not zero.

Hello, PC World(2)

The above can be assembled (the syntax is NASM's) to give a remarkably short `.COM` file: just 32 bytes.

A disassembler interprets the resulting file as follows

```
D:\MJR\ASM\NASM>debug hello1.com
-u
0C80:0100 B800B8        MOV     AX,B800
0C80:0103 8EC0          MOV     ES,AX
0C80:0105 BFA000        MOV     DI,00A0
0C80:0108 8D361401      LEA     SI,[0114]
0C80:010C B90C00        MOV     CX,000C
0C80:010F A4            MOVSB
0C80:0110 47            INC     DI
0C80:0111 E2FC          LOOP    010F
0C80:0113 C3            RET
```

Hello, PC World(3)

The `.COM` file has been loaded at into segment number 0x0c80 at the expected offset of 0x100.

Note the variable instruction lengths, one to four bytes here, and the backward (little-endian) nature of the storage of immediate data: A000 for 00A0 (160), 1401 for 0114, 0C00 for 000C (12), etc. The reference to 0114 is to the string “Hello, World” which follows the executable instructions immediately.

In the loop instruction, the jump is -4 bytes, as the next instruction will be 10F (again) not 113. Converting -4 to two’s complement, one gets FC. Label names have been lost – there is no occurance of ‘next_ch’ or ‘string’ in the `.COM` file.

Calling DOS

The above code has several disadvantages. It works in just one video mode. It always writes at the same location on the screen, regardless of what was there. It requires precise knowledge of the hardware. Its output does not obey the normal redirections (’>’ and ’`|`’).

DOS provides a function for writing a string to the terminal, which works in whichever video mode is in use, which writes at the current cursor position, and which does obey redirections. DOS is called via the CPU’s ‘interrupt’ instruction, normally interrupt number 0x21. The arguments to the function are passed in the CPU’s registers. Most importantly, the `ah` register specifies which DOS function one requires.

Function 9 prints a string from the address in `dx`. The string must be terminated by a ‘`$`’.

Function 0x4C exits, returning the contents of `al` as the exit code.

Hello, DOS World

```
section .text
org 0x100

  lea dx,[string]
  mov ah,9
  int 0x21

  mov ah,0x4C
  mov al,0
  int 0x21

string db "Hello, World$"
```

Now a mere 27 bytes!

Real Operating Systems: Linux

A real operating system would not allow direct hardware access as used in the first example above (indeed, in the presence of virtual addressing, the first example is nonsensical). It would insist on a coding style like the second.

However, like DOS it is called via an interrupt instruction, and again the required function and its arguments are placed in the CPU registers. Unlike DOS, the interrupt is always number 0x80.

Being C-based, UNIX tends to have functions similar to some of the C functions. Two of interest here are `write()` and `exit()`. In Linux `write` is function number four, and has three arguments: file descriptor, pointer to data, and length of data to be written.

Linux uses a more structured form for its binary files, called ELF.

(ELF = Enhanced Library Format.)

Hello, Linux World

```
section .text
global _start
_start
  mov eax,4      ; write is function 4
  mov ebx,1      ; unit 1 is stdout
  lea ecx,[msg] ; pointer to message
  mov edx,13     ; length of message
  int 0x80
  mov eax,1      ; exit is function 1
  mov ebx,0      ; exit code of zero
  int 0x80

  msg db "Hello, World",10
```


Hello, Linux World (2)

This file can be run (on a 32 bit Linux PC) using:

```
$ nasm -f elf  hello.asm
$ ld hello.o
$ ./a.out
Hello, World
```

Here `ld` is not linking, merely adding (more) ELF magic to the object file. It likes the global symbol `_start` to specify where execution should commence. Some superfluous information can be removed with the command `strip a.out`.

The resulting binary file is 364 bytes long. That it contains more structure than the DOS `.COM` file can be revealed by

```
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386,
 version 1 (SYSV), statically linked, stripped
```

Babel

Those used to the more command AT&T / Gnu assembler syntax will have been surprised by these examples which are in Intel's syntax. They would prefer:

```
.section .text
.global _start
_start:
   movl $4,%eax           # write is function 4
   movl $1,%ebx           # unit 1 is stdout
   lea msg,%ecx           # pointer to message
   movl $12,%edx          # length of message
   int $0x80
   movl $1,%eax           # exit is function 1
   movl $0,%ebx           # exit code of zero
   int $0x80

   msg:
     .ascii "Hello World\n"
```

Note the dollars before constants, %s before register names, the reversal of the order of the operands, and the change of the comment character. Create a binary with:

```
 as -o hello.o hello.s ; ld hello.o
```


The `int` in detail

In the DOS example, we chose to call DOS via the conventional `int 0x21` call. However, the DOS 'kernel' ran with the same privileges as our own code, and we could have jumped into it by any route. Executing `int 0x21` merely places a return address on the stack, and jumps to the address given by entry number 0x21 in the interrupt vector table, which, for the 8086, starts at address zero, occupies the first 1K of memory, and is easily read or modified.

In Linux, `int 0x80` is rather different. The address it refers to is not modifiable by the user code, and when it is executed, a flag in the CPU is immediately set to indicate that the CPU is executing kernel code. When this flag is set, direct hardware access is possible. The flag gets reset as the execution returns to the original program. Any attempt to execute the privileged instructions which the kernel uses without this flag set will be denied by the CPU itself. There is a very clear distinction between 'kernel space' and 'user space'.

The 8086 interrupt table has 256 four-byte (segment then offset) entries.

The IA32 processors have several modes of operation. The default, used by DOS, has no concept of privileged instructions – everything is always acceptable. The modes which Linux (MacOS X, Windows) use do enforce different privilege levels.

Using libraries

As a first example of using a library, we shall convert the above Linux code to call the `write()` and `_exit()` functions from `libc`, rather than using the kernel interface directly.

The most important UNIX library, `libc`, contains all the (non-maths) functions required by ANSI C and any extensions supported by the platform, as well as C wrappers to all kernel calls. Thus some of its functions, such as `strlen()`, do not call the kernel at all, some, such as `printf()` do considerable work before calling a more basic kernel function and others, such as `write()`, are trivial wrappers for kernel functions.

The last category is traditionally documented in section 2 of the manual pages, whereas the others are in section 3.

Some C functions call kernel functions occasionally, such as `malloc()`, which needs to provide any amount of memory that the program requests, but can only request memory from the kernel in multiples of the page size (typically 4K or 8K).

Yes, UNIX has an online manual for C functions. E.g. '`man 3 printf`' for those who cannot remember ever format specifier.

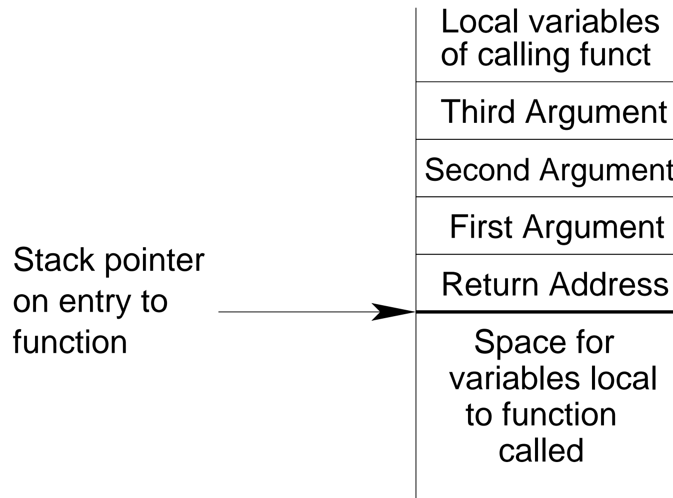
Using libraries: 2

Several changes are necessary to our code. The symbols `write` and `_exit` need to be declared to the assembler as *external* – they are not defined in the code given, and the linker will insert the relevant code.

The routines in `libc` can be invoked using the `call` instruction, but they do not expect their arguments to be in registers, but rather on the stack. The stack is a downwards-growing area of scratch memory whose next free address is given by the register `esp`. An instruction such as `push eax` puts a copy of the value in `eax` on the stack, and subtracts four from `esp` (as four bytes are needed to store the value in `eax`).

The `call` instruction also uses the stack for storing the *return address* – the address to return to after the function exits.

A Stack of Arguments



The stack mixes function return addresses, local variables and arguments in close proximity, with predictable, bad, results. It may also contain compiler-generated temporary variables.

Here all the arguments were of the same size. Of course double precision numbers, and, on 64 bit machines, pointers, will be twice the size of a 32 bit integer. The use of the stack for *all* arguments is inefficient. Sometimes the first few arguments are sent in registers instead. But of course the library and the calling program must agree about how arguments are transfered!

In practice, all C compilers on Linux agree, and for this, and other, reasons, one can compile different bits of a C program with different compilers, and everything works. Fortran compilers agree about almost nothing. This is why one needs a separate NAG (and MKL and MPI and . . .) library for the Intel, PathScale, Gnu, Portland, Sun, . . . , compiler, and it is very tedious.

Hello, libc World

```
.section .text
.extern write
.extern _exit
.global _start
_start:
  movl $13,%eax # length of message
  push %eax
  lea msg,%eax  # pointer to message
  push %eax
  movl $1,%eax  # unit 1 is stdout
  push %eax
  call write    # write(fd,*buff,count)
  pop %eax      # remove the three arguments
  pop %eax
  pop %eax
  mov $0,%eax
  push %eax
  call _exit    # _exit(status)

  msg:
    .ascii "Hello, World\n"
```

The Linker

The above code can be compiled and linked with

```
$ as -o hello.libc.o hello.libc.asm
$ ld -static hello.libc.o -lc
```

The assembler was told that `write` and `exit` were external symbols, that is, symbols used but not defined in the source file.

Linkers join together collections of binary object files resolving such references. A library is merely a collection of many object files from separate source files gathered into a single archive file. The library `libc.a`, which will contain a superset of those functions required by the ANSI C standard, on the system used was a 2.5MB archive formed from 1,300 separate .o files.

The linker extracted just those routines required and placed them in the resulting executable. So the final executable file was 1770 bytes.

Don't try this at home! A modern libc expects various initialisation routines to be called before it is used, and will simply segfault in this example.

Linker and Compiler command lines

The option `-lfoo` is simply shorthand for 'look for `libfoo.a` in all the directories where library files are expected, included those directories specified by `-L` options.' So specifying `-lc` here is equivalent to specifiying `/usr/lib/libc.a`.

Computers read from left to right.

```
$ ld -static -lc hello.libc.o
```

fails reporting undefined references to `write` and `_exit`. Initially the linker regards `_start` as its only undefined symbol. In `libc.a` it found no definition of `_start`, so included none of that archive. Looking in `hello.lib.c.o` it found a definition of `_start`, but gained `write` and `_exit` as unresolved symbols. Continuing through the files in the order given, there are no more left from which to find definitions of these symbols.

Fortunately modern compilers often do work if one specifies libraries before the source files which require them. It means that well-educated Humans can detect which of their colleagues are less than precise in their approach to their work.

FIX ME!

It should be clear that none of the sections taken from the library will end up at any particular address. Their destination will depend on the size of the user-supplied program, and which other library functions have been included.

The linker performs the task of relocating the code, 'fixing' any absolute addresses within the library code (text and data) as required.

The virtual addresses at which the text, initialised data and uninitialised data (the fixed-sized segments) will be loaded is fixed at link time.

Moving to C

A C programmer would probably use the function `printf()` rather than `write()` in a 'Hello World' program. A C compiler expects the start of the code to be a function called `main`, not a point called `_start`.

In C `main` is a function returning an integer, and for 32 bit Linux this means that the return value should be placed in the register eax.

Strings in C are terminated with a null byte (rather than the Fortran / Pascal practice of storing a separate length, and having no byte as a special end-of-string marker).

Hello, mostly C

```
.section .text
.extern printf
.global main
main:
  lea msg,%eax # pointer to message
  push %eax
  call printf  # printf(*buff)
  pop %eax     # remove one argument
  movl $0,%eax
  ret          # return from function

msg:
  .ascii "Hello, World\n"
  .byte 0
```

This needs compiling as

```
$ as -o hello.c.o hello.c.s
$ gcc -static hello.c.o
```

And this might even work on a current 32-bit Linux installation.

`gcc` vs `ld`

This was not linked with `ld`, but with `gcc`. What is the difference?

Firstly `gcc` assumes a `-lc` automatically.

Secondly, to cope with the fact that a C programmer expects the starting point to be an integer function called `main` (taking two or three arguments, which we are ignoring), and UNIX expects the starting point to be called `_start`, `gcc` includes a tiny object file which has an entry point called `_start` and itself calls `main`. This file is called `crt1.o` by `gcc`.

In practice `gcc` includes a few other tiny object files, and sometimes the odd library too. And, indeed, `crt1.o` actually calls `__libc_start_main` so that `libc` can do any initialisation it wishes, and it then calls `main`.

The days when one could call `ld` directly to link are probably over for those who wish to remain sane.

Of course, had `gcc` been offered a file ending `.c` rather than `.o`, it would have compiled it first and passed the resulting `.o` file to the linker.

Bloat

The problem with our `gcc`-linked program is that it is 2.6MB (on a machine with a 15MB `libc.a`). Even a completely null program is this length.

The reason people have ceased caring about such bloat is that executables today usually perform dynamic linking (or runtime linking). The linker merely places in the executable the names of the libraries which will be needed at runtime, and checks that they do resolve all unresolved symbols. The actual linking process occurs every time the program is run.

Dynamic linking is the default. The `-static` options above changed back to the older style of linking.

Unfortunately one can not use the same library for both static and dynamic linking. So every library appears twice. Once, as a `.a` file for static linking (and only needed to support compiling in conjunction with static linking), and once as a `.so` file (shared object) which is needed at both compile time and run time for dynamic executables.

Dynamic or Static?

```
$ gcc -static hello.C.o
```

produces a 2.6MB executable, whereas

```
$ gcc hello.C.o
```

produces a 9.7KB executable. One can tell which libraries are being used at run-time by typing

```
$ ldd ./a.out
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb76ec000)
        /lib/ld-linux.so.2 (0xb786d000)
```

(The first and the last of the above are needed by every dynamic executable running on Linux.)

Libraries and Paths

At compile time, the linker searches a set of default directories for libraries, probably
/lib, /usr/lib, and maybe /usr/local/lib
but maybe with lib replaced by lib64. At run-time the dynamic linker searches a set of directories which is probably set to be the same.

The linker option '-L' prefixes directories to the standard search path at compile time. The linker option '-rpath' does the same for the run-time linkage, storing the extra path in the executable. The environment variable LD_LIBRARY_PATH, if set, is also used at run-time for finding libraries.

So, if linking against a library which is in an unusual place, there are three choices:

1/ Make sure you link against a static library
2/ If the library is shared, and unlikely to move, use -L and -rpath.
3/ If the library is likely to move (e.g. you might try giving a bundle of files to a friend), then LD_LIBRARY_PATH may be needed.

(The rules for shared libraries which require additional shared libraries which were not specified at compile time are different and confusing. Once one has worked out which libraries they will require, specifying them at compile time solves all problems.)

Dynamic Advantages

If each of the 2,000 executables residing in /usr/bin needed to be just 2MB bigger, the disk space requirement would be annoying. However, X11 programs tend to be dynamically linked against tens of MB of libraries, so the increase could be more dramatic.

Also significant is the gain in physically memory use. If two programs use the same dynamic library, it is loaded just once into physical memory, even if different users are executing it. If they are statically linked, the kernel has no way of telling which parts of the executables are common, and two copies are needed in physical memory.

If a bug is fixed in a dynamic library, any program using it will get the new version next time it is run, with no need to relink or recompile.

Dynamic Disadvantages

Programs are no longer stand-alone binary blobs, but need a host of libraries, of the correct version, on the machine on which they are run. Users have some control over this via LD_LIBRARY_PATH.

Beware that LD_LIBRARY_PATH is searched first every time any dynamic executable is run. Point this at an NFS-mounted directory, and if there is any problem on that server even simple commands such as ls on a local directory will take forever, as ls is a dynamic executable, so will attempt to use the target of LD_LIBRARY_PATH for finding its libraries before looking at the standard system libraries.

If a bug is introduced in a dynamic library, programs using it will get the new version next time it is run, and programs which used to work cease working. So also the same program can give different answers on different computers.

If one links against a library unnecessarily (no symbols from that library were actually required), its presence is still required at run-time. So don't link everything against everything just in case. Only Gnomes do that.

There is a (very) slight and often overstated overhead on every function call.

Not C

There are two approaches for compilers of non-C languages. One could call the kernel directly from some sort of libc equivalent designed for the language in question. This is tedious compared to having a language-specific library which, rather than calling the kernel, uses libc for access to kernel functions.

So most Fortran compilers have a Fortran library, often called libf, which supplies those functions which a Fortran programmer expects, and, when kernel assistance is required (e.g. for write and open, but not for sqrt or sin) calls libc.

Avoiding Collisions

If one is linking against `libc`, one must avoid using the name of any C function in one's code. This is fine in C, but it is unreasonable to expect Fortran programmers to avoid using names such as `rand`, `abort`, `system`, `qsort` etc. because C got there first.

The solution used by most Fortran compilers is to append an underscore to all external names. No external symbol in the C library ends in an underscore, so collisions are avoided. Calling functions in `libc` directly from Fortran becomes impossible, as any attempt to write

```
call qsort(...)
```

will look for a routine called `qsort_`. However, only the very simplistic would think that they had a hope of working out the syntax to get Fortran to pass the four arguments which `qsort` required: a pointer to an array, two `size_t` objects, and a pointer to a function.

Overloading

Neither the C language nor `ld` permit function overloading, that is, functions behaving differently depending on the number and type of their arguments. C++ and Fortran 90 do. When a C++ programmer writes

```
int add(int a, int b);
double add(double a, double b);

void dummy(int x, double y){
  *x=add(x,x);
  *y=add(y,y);
}
```

the linker needs to see three distinct functions. With C++ on Linux these names are *mangled* to `_Z3adddd`, `_Z3addii` and `_Z5dummyid`. As C/C++ identifiers must start with a letter, there is no ambiguity here. A debugger which understands C++ will demangle the names automatically. The same situation applies to Fortran 90.

A C++ program calling (or creating) a C function must do so explicitly.

```
extern "C" void dummy(int x, double y){ ...
```

Debugging

Names which are not needed by the dynamic linker are not needed in an executable. So a static executable does not need to retain any hint of the variable names or function names which were in the original program. A dynamic executable does need to retain the names of those functions which it calls from dynamic libraries, but that is all.

The inclusion of original variable and function names can be extremely useful when debugging. By default most compilers include function name information in their code, and some variable names too. If encouraged with options such as `-g`, the names of all local variables are included, and even which source line number gave rise to which instructions (a slightly ill-defined concept, particularly on high optimisation levels).

Including detailed debugging information can easily double the size of the executable. It need not slow it down, although `-g` sometimes implies `-O0` to prevent the compiler from optimising variables entirely into registers, reordering lines, etc.

Debugging the Stack

An executable with debugging information will contain information about which address ranges correspond to which of its functions. If a program crashes and produces a ‘core dump’, that dump contains a snapshot of the program’s memory and registers at the time the crash occured.

A debugger can examine the dump, work out what function was active when the dump happened (from the value of the instruction pointer register), and, making use of the extra ‘base pointer’ register, it can walk backwards through the stack to find the complete call tree until it reaches the initial `main()`’ function.

This works perfectly, *provided the stack is not corrupted*. However, programs often crash precisely because they have suffered from variables over-writing each other, at which point the vulnerable structure of ‘frames’ on the stack will be destroyed, and the debugger will have no clue where it is.

Really bad debuggers then crash themselves. In the worse cases, they produce core dumps which over-write the one which you were attempting to analyse. Yes, such lunacy has persisted into the 21st century.

Compilers & Optimisation

Optimisation

Optimisation is the process of producing a machine code representation of a program which will run as fast as possible. It is a job shared by the compiler and programmer.

The compiler uses the sort of highly artificial intelligence that programs have. This involves following simple rules without getting bored halfway through.

The human will be bored before he starts to program, and will never have followed a rule in his life. However, it is he who has the Creative Spirit.

This section discussed some of the techniques and terminology used.

Loops

Loops are the only things worth optimising. A code sequence which is executed just once will not take as long to run as it took to write. A loop, which may be executed many, many millions of times, is rather different.

```
do i=1,n
  x(i)=2*pi*i/k1
  y(i)=2*pi*i/k2
enddo
```

Is the simple example we will consider first, and Fortran will be used to demonstrate the sort of transforms the compiler will make during the translation to machine code.

Simple and automatic

CSE

```
do i=1,n
  t1=2*pi*i
  x(i)=t1/k1
  y(i)=t1/k2
enddo
```

Common Subexpression Elimination. Rely on the compiler to do this.

Invariant removal

```
t2=2*pi
do i=1,n
  t1=t2*i
  x(i)=t1/k1
  y(i)=t1/k2
enddo
```

Rely on the compiler to do this.

Division to multiplication

```
t2=2*pi
t3=1/k1
t4=1/k2
do i=1,n
  t1=t2*i
  x(i)=t1*t3
  y(i)=t1*t4
enddo
```

```
t1=2*pi/k1
t2=2*pi/k2
do i=1,n

  x(i)=i*t1
  y(i)=i*t2
enddo
```

```
t1=2*pi/k1
t2=2*pi/k2
do i=1,n
  t=real(i,kind(1d0))
  x(i)=t*t1
  y(i)=t*t2
enddo
```

From left to right, increasingly optimised versions of the loop after the elimination of the division.

The compiler shouldn't default to this, as it breaks the IEEE standard subtly. However, there will be a compiler flag to make this happen: find it and use it!

Conversion of `x**2` to `x*x` will be automatic.

Remember multiplication is many times faster than division, and many many times faster than logs and exponentiation.

Some compilers now do this by default, defaulting to breaking IEEE standards for arithmetic. I prefered the more Conservative world in which I spent my youth.

Another example

```
  y=0
  do i=1,n
    y=y+x(i)*x(i)
  enddo
```

As machine code has no real concept of a loop, this will need converting to a form such as

```
  y=0
  i=1
1   y=y+x(i)*x(i)
    i=i+1
    if (i<n) goto 1
```

At first glance the loop had one fp add, one fp multiply, and one fp load. It also had one integer add, one integer comparison and one conditional branch. Unless the processor supports speculative loads, the loading of `x(i+1)` cannot start until the comparison completes.

Unrolling

```
y=0
do i=1,n-mod(n,2),2
  y=y+x(i)*x(i)+x(i+1)*x(i+1)
enddo
if (mod(n,2)==1) y=y+x(n)*x(n)
```

This now looks like

```
  y=0
  i=1
  n2=n-mod(n,2)
1   y=y+x(i)*x(i)+x(i+1)*x(i+1)
    i=i+2
    if (i<n2) goto 1
  if (mod(n,2)==1) y=y+x(n)*x(n)
```

The same 'loop overhead' of integer control instructions now deals with two iterations, and a small *coda* has been added to deal with odd loop counts. Rely on the compiler to do this.

The compiler will happily unroll to greater *depths* (2 here, often 4 or 8 in practice), and may be able to predict the optimum depth better than a human, because it is processor-specific.

Reduction

This dot-product loop has a nasty data dependency on `y`: no add may start until the preceeding add has completed. However, this can be improved:

```
t1=0 ; t2=0
do i=1,n-mod(n,2),2
  t1=t1+x(i)*x(i)
  t2=t2+x(i+1)*x(i+1)
enddo
y=t1+t2
if (mod(n,2)==1) y=y+x(n)*x(n)
```

There are no data dependencies between `t1` and `t2`. Again, rely on the compiler to do this.

This class of operations are called reduction operations for a 1-D object (a vector) is reduced to a scalar. The same sort of transform works for the sum or product of the elements, and finding the maximum or minimum element.

Reductions change the order of arithmetic operations and thus change the answer. Conservative compilers won't do this without encouragement.

Again one should rely on the compiler to do this transformation, because the number of partial sums needed on a modern processor for peak performance could be quite large, and you don't want your source code to become an unreadable lengthy mess which is optimised for one specific CPU.

Prefetching

```
  y=0
  do i=1,n
    prefetch_to_cache x(i+8)
    y=y+x(i)*x(i)
  enddo
```

As neither C/C++ nor Fortran has a prefetch instruction in its standard, and not all CPUs support prefetching, one must rely on the compiler for this.

This works better after unrolling too, as only one prefetch per cache line is required. Determining how far ahead one should prefetch is awkward and processor-dependent.

It is possible to add directives to one's code to assist a particular compiler to get prefetching right: something for the desperate only.

Loop Elimination

```
do i=1,3
  a(i)=0
endo
```

will be transformed to

```
a(1)=0
a(2)=0
a(3)=0
```

Note this can only happen if the iteration count is small *and* known at compile time. Replacing '3' by 'n' will cause the compiler to unroll the loop about 8 times, and will produce dire performance if n is always 3.

Loop Fusion

```
do i=1,n
  x(i)=i
enddo
do i=1,n
  y(i)=i
enddo
```

transforms trivially to

```
do i=1,n
  x(i)=i
  y(i)=i
enddo
```

eliminating loop overheads, and increasing scope for CSE. Good compilers can cope with this, a few cannot.

Assuming `x` and `y` are real, the implicit conversion of `i` from integer to real is a common operation which can be eliminated.

Fusion or Fission?

Ideally temporary values within the body of a loop, including pointers, values accumulating sums, etc., are stored in registers, and not read in and out on each iteration of the loop. A sane RISC CPU tends to have 32 general-pupose integer registers and 32 floating point registers.

Intel's 64 bit processors have just 16 integer registers, and 16 floating point vector registers storing two (or four in recent processors) values each. Code compiled for Intel's 32 bit processors uses just half this number of registers.

A 'register spill' occurs when a value which ideally would be kept in a register has to be written out to memory, and read in later, due to a shortage of registers. In rare cases, loop fission, splitting a loop into two, is preferable to avoid a spill.

Fission may also help hardware prefetchers spot memory access patterns.

Strength reduction

```
double a(2000,2000)

do j=1,n
  do i=1,n
    a(i,j)=x(i)*y(j)
  enddo
enddo
```

The problem here is finding where the element `a(i,j)` is in memory. The answer is $8(i-1)+16000(j-1)$ bytes beyond the first element of `a`: a hideously complicated expression.

Just adding eight to a pointer every time `i` increments in the inner loop is much faster, and called strength reduction. Rely on the compiler again.

Inlining

```
function norm(x)
double precision norm,x(3)

norm=x(1)**2+x(2)**2+x(3)**2
end function
...
a=norm(b)
```

transforms to

```
a=b(1)**2+b(2)**2+b(3)**2
```

eliminating the overhead of the function call.

Often only possible if the function and caller are compiled simultaneously.

Instruction scheduling and loop pipelining

A compiler ought to move instructions around, taking care not to change the resulting effect, in order to make best use of the CPU. It needs to ensure that latencies are 'hidden' by moving instructions with data dependencies on each other apart, and that as many instructions as possible can be done at once. This analysis is most simply applied to a single pass through a piece of code, and is called *code scheduling*.

With a loop, it is unnecessary to produce a set of instructions which do not do any processing of iteration `n+1` until all instructions relating to iteration `n` have finished. It may be better to start iteration `n+1` before iteration `n` has fully completed. Such an optimisation is called *loop pipelining* for obvious reasons..

Sun calls 'loop pipelining' 'modulo scheduling'.

Consider a piece of code containing three integer adds and three fp adds, all independent. Offered in that order to a CPU capable of one integer and one fp instruction per cycle, this would probably take five cycles to issue. If reordered as $3\times$(integer add, fp add), it would take just three cycles.

Debugging

The above optimisations should really never be done manually. A decade ago it might have been necessary. Now it has no beneficial effect, and makes code longer, less readable, and harder for the compiler to optimise!

However, one should be aware of the above optimisations, for they help to explain why line-numbers and variables reported by debuggers may not correspond closely to the original code. Compiling with all optimisation off is occassionally useful when debugging so that the above transformations do not occur.

Loop interchange

The conversion of

```
do i=1,n
  do j=1,n
    a(i,j)=0
  enddo
enddo
```

to

```
do j=1,n
  do i=1,n
    a(i,j)=0
  enddo
enddo
```

is one loop transformation most compilers do get right. There is still no excuse for writing the first version though.

The Compilers

```
f90 -fast -o myprog myprog.f90 func.o -lnag
```

That is options, source file for main program, other source files, other objects, libraries. Order does matter (to different extents with different compilers), and should not be done randomly.

Yet worse, random options whose function one cannot explain and which were dropped from the compiler's documentation two major releases ago should not occur at all!

The compile line is read from left to right. Trying

```
f90 -o myprog myprog.f90 func.o -lnag -fast
```

may well apply optimisation to nothing (i.e. to the source files following `-fast`). Similarly

```
f90 -o myprog myprog.f90 func.o -lnag -lcxml
```

will probably use routines from NAG rather than cxml if both contain the same routine. However,

```
f90 -o myprog -lcxml myprog.f90 func.o -lnag
```

may also favour NAG over cxml with some compilers.

Calling Compilers

Almost all UNIX commands never care about file names or extensions.

Compilers are very different. They do care greatly about file names, and they often use a strict left to right ordering of options.

Extension	File type
.a	static library
.c	C
.cc	C++
.cxx	C++
.C	C++
.f	Fixed format Fortran
.F	ditto, preprocess with cpp
.f90	Free format Fortran
.F90	ditto, preprocess with cpp
.i	C, do not preprocess
.o	object file
.s	assembler file

Consistency

It is usual to compile large programs by first compiling each separate source file to an object file, and then linking them together.

One must ensure that one's compilation options are consistent. In particular, one cannot compile some files in 32 bit mode, and others in 64 bit mode. It may not be possible to mix compilers either: certainly on our Linux machines one cannot link together things compiled with NAG's f95 compiler and Intel's ifc compiler.

Common compiler options

`-lfoo` and `-L`

`-lfoo` will look first for a shared library called `libfoo.so`, then a static library called `libfoo.a`, using a particular search path. One can add to the search path (`-L${HOME}/lib` or `-L.`) or specify a library explicitly like an object file, e.g. `/temp/libfoo.a`.

`-O`, `-O`**n** and `-fast`

Specify optimisation level, `-O0` being no optimisation. What happens at each level is compiler-dependent, and which level is achieved by not specifying `-O` at all, or just `-O` with no explicit level, is also compiler dependent. `-fast` requests fairly aggressive optimisation, including some unsafe but probably safe options, and probably tunes for specific processor used for the compile.

`-c` and `-S`

Compile to object file (`-c`) or assembler listing (`-S`): do not link.

`-g`

Include information about line numbers and variable names in `.o` file. Allows a debugger to be more friendly, and may turn off optimisation.

More compiler options

`-C`

Attempt to check array bounds on every array reference. Makes code much slower, but can catch some bugs. Fortran only.

`-r8`

The `-r8` option is entertaining: it promotes all single precision variables, constants and functions to double precision. Its use is unnecessary: code should not contain single precision arithmetic unless it was written for a certain Cray compiler which has been dead for years. So your code should give identical results whether compiled with this flag or not.

Does it? If not, you have a lurking reference to single precision arithmetic.

The rest

Options will exist for tuning for specific processors, warning about unused variables, reducing (slightly) the accuracy of maths to increase speed, aligning variables, etc. There is no standard for these.

IBM's equivalent of `-r8` is `-qautodbl=dbl4`.

A Compiler's view: Basic Blocks

A compiler will break source code into *basic blocks*. A basic block is a sequence of instructions with a single entry point and a single exit point. If any instruction in the sequence is executed, all must be executed precisely once.

Some statements result in multiple basic blocks. An if/then/else instruction will have (at least) three: the conditional expression, the then clause, and the else clause. The body of a simple loop may be a single basic block, provided that it contains no function calls or conditional statements.

Compilers can amuse themselves re-ordering instructions within a basic block (subject to a little care about dependencies). This may result in a slightly complicated correspondence between line numbers in the original source code and instructions in the compiled code. In turn, this makes debugging more exciting.

A Compiler's view: Sequence Points

A sequence point is a point in the source such that the consequences of everything before it point are completed before anything after it is executed. In any sane language the end of a statement is a sequence point, so

```
a=a+2
a=a*3
```

is unambiguous and equivalent to `a=(a+2)*3`.

Sequence points usually confuse C programmers, because the increment and decrement operators `++` and `--` do not introduce one, nor do the commas between function arguments.

```
j=(++i)*2+(++i);
printf("%d %d %d\n",++i,++i,++i);
```

could both do *anything*. With `i=3`, the first produces 13 with most compilers, but 15 with Open64 and PathScale. With `i=5`, the latter produces '`6 7 8`' with Intel's C compiler and '`8 8 8`' with Gnu's. Neither is wrong, for the subsequent behaviour of the code is completely undefined according to the C standard. No compiler tested produced a warning by default for this code.

And: there's more

```
if ((i>0)&&(1000/i)>1) ...
```

```
if ((i>0).and.(1000/i>1)) ...
```

The first line is valid, sane, C. In C `&&` is a sequence point, and logical operators guarantee to short-circuit. So in the expression

```
A&&B
```

A will be evaluated before B, and if A is false, B will not be evaluated at all.

In Fortran none of the above is true, and the code may fail with a division by zero error if `i=0`.

```
A.and.B
```

makes no guarantees about evaluation order, or in what circumstances both expressions will be evaluated.

What is true for `&&` in C is also true for `||` in C.

Fortran 90

Fortran 90 is *the* language for numerical computation. However, it is not perfect. In the next few slides are described some of its many imperfections.

Lest those using C, C++ and Mathematica feel they can laugh at this point, nearly everything that follows applies equally to C++ and Mathematica. The only (almost completely) safe language is F77, but that has other problems.

Most of F90's problems stem from its friendly high-level way of handling arrays and similar objects.

So that I am not accused of bias,

`http://www.tcm.phy.cam.ac.uk/~mjr/C/`

discusses why C is even worse. . .

Slow arrays

```
a=b+c
```

Humans do not give such a simple statement a second glace, quite forgetting that depending what those variables are, that could be an element-wise addition of arrays of several million elements. If so

```
do i=1,n
  a(i)=b(i)+c(i)
enddo
```

would confuse humans less, even though the first form is neater. Will both be treated equally by the compiler? They should be, but many early F90 compilers produce faster code for the second form.

Big surprises

```
a=b+c+d
```

really ought to be treated equivalently to

```
do i=1,n
  a(i)=b(i)+c(i)+d(i)
enddo
```

if all are vectors. Many early compilers would instead treat this as

```
temp_allocate(t(n))
do i=1,n
  t(i)=b(i)+c(i)
enddo
do i=1,n
  a(i)=t(i)+d(i)
enddo
```

This uses much more memory than the F77 form, and is much slower.

Sure surprises

```
a=matmul(b,matmul(c,d))
```

will be treated as

```
temp_allocate(t(n,n))
t=matmul(c,d)
a=matmul(b,t)
```

which uses more memory than one may first expect. And is the `matmul` the compiler uses as good as the `matmul` in the BLAS library? Not if it is Compaq's compiler.

I don't think Compaq is alone in being guilty of this stupidity. See IBM's `-qessl=yes` option. . .

Note that even `a=matmul(a,b)` needs a temporary array. The special case which does not is `a=matmul(b,c)`.

Slow Traces

```
allocate (a(16384,16384))

call tr(a(1:nn,1:nn),nn,x)

subroutine tr(m,n,t)
double precision m(n,n),t
integer i,n

t=0
do i=1,n
  t=t+m(i,i)
enddo

end subroutine
```

As `n` was increased by factors of two from 512 to 16384, the time in seconds to perform the trace was 3ms, 13ms, 50ms, 0.2s, 0.8s, 2ms.

Mixed Languages

The `tr` subroutine was written in perfectly reasonable Fortran 77. The call is perfectly reasonable Fortran 90. The mix is not reasonable.

The subroutine requires that the array it is passed is a contiguous 2D array. When `n=1024` it requires `m(i,j)` to be stored at an offset of $8(i-1)+8192(j-1)$ from `m(1,1)`. The original layout of `a` in the calling routine of course has the offsets as $8(i-1)+131072(j-1)$.

The compiler must create a new, temporary array of the shape which `tr` expects, copy the relevant part of `a` into, and, after the call, copy it back, because in general a subroutine may alter any elements of any array it is passed.

Calculating a trace should be order n in time, and take no extra memory. This poor coding results in order n^2 in time, *and* n^2 in memory.

In the special case of `n=16384` the compiler notices that the copy is unnecessary, as the original is the correct shape.

Bright people deliberate limit their stack sizes to a few MB (see the output of `ulimit -s`. Why? As soon as their compiler creates a large temporary array on the stack, their program will segfault, and they are thus warned that there is a performance issue which needs addressing.

Pure F90

```
use magic

call tr(a(1:nn,1:nn),nn,x)

module magic
contains
subroutine tr(m,n,t)
double precision m(:,:),t
integer i,n

t=0
do i=1,n
  t=t+m(i,i)
enddo

end subroutine
end module magic
```

This is decently fast, and does not make extra copies of the array.

Pure F77

```
allocate (a(16384,16384))

call tr(a,16384,nn,x)

subroutine tr(m,msize,n,t)
double precision m(msize,msize),t
integer i,n,msize

t=0
do i=1,n
  t=t+m(i,i)
enddo

end subroutine
```

That is how a pure F77 programmer would have written this. It is as fast as the pure F90 method (arguably marginally faster).

Type trouble

```
type electron
   integer :: spin
   real (kind(1d0)), dimension(3) :: x
end type electron

type(electron), allocatable :: e(:)
allocate (e(10000))
```

Good if one always wants the spin and position of the electron together. However, counting the net spin of this array

```
s=0
do i=1,n
  s=s+e(i)%spin
enddo
```

is now slow, as an electron will contain 4 bytes of spin, 4 bytes of padding, and three 8 byte doubles, so using a separate spin array so that memory access was unit stride again could be eight times faster.

What is temp_allocate?

Ideally, an allocate and deallocate if the object is 'large', and placed on the stack otherwise, as stack allocation is faster, but stacks are small and never shrink. Ideally reused as well.

```
a=matmul(a,b)
c=matmul(c,d)
```

should look like

```
temp_allocate(t(n,n))
t=matmul(a,b)
a=t
temp_deallocate(t)
temp_allocate(t(m,m))
t=matmul(c,d)
c=t
temp_deallocate(t)
```

with further optimisation if m=n. Some early F90 compilers would allocate all temporaries at the beginning of a subroutine, use each once only, and deallocate them at the end.

a=sum(x*x)

```
temp_allocate(t(n))
do i=1,n
  t(i)=x(i)*x(i)
enddo
a=0
do i=1,n
  a=a+t(i)
enddo
```

or

```
a=0
do i=1,n
  a=a+x(i)*x(i)
enddo
```

Same number of arithmetic operations, but the first has $2n$ reads and n writes to memory, the second n reads and no writes (assuming a is held in a register in both cases). Use a=dot_product(x,x) not a=sum(x*x)! Note that a compiler good at loop fusion may rescue this code.

Universality

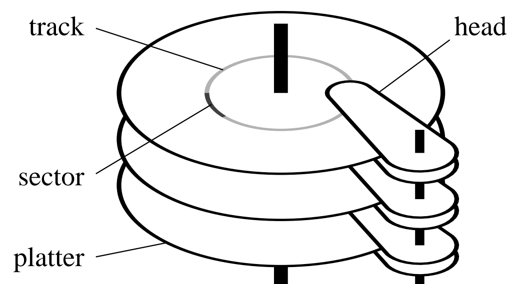
The above examples pick holes in Fortran 90's array operations. This is not an attack on F90 – its array syntax is very convenient for scientific programming. It is a warning that applies to all languages which support this type of syntax, including Matlab, Python, C++ with suitable overloading, etc.

It is not to say that all languages get all examples wrong. It is to say that most languages get some examples wrong, and, in terms of efficiency, wrong can easily cost a factor of two in time, and a large block of memory too. Whether something is correctly optimised may well depend on the precise version of the compiler / interpreter used.

Disks, Filesystem & Fileservers

A Physical Disk Drive

A single hard disk contains a spindle with multiple *platters*. Each platter has two magnetic surfaces, and at least one head 'flying' over each surface. The heads do fly, using aerodynamic effects in a dust-free atmosphere to maintain a very low altitude. Head crashes (head touching surface) are catastrophic. There is a special 'landing zone' at the edge of the disk where the heads must settle when the disk stops spinning.



Disk Drives vs Memory

Memory and disk drives differ in some important respects.

Disk drives retain data in the absence of power.

Memory is addressable at the level of bytes, disk drives at the level of blocks, typically 512 bytes.

Disk drives are cheaper per unit of storage. Disks cost around £40 per TB, and a typical PC case can contain about 10TB of disks, memory costs around £6,000 per TB, and a typical PC case can contain about 0.03TB.

Disk Drives vs Memory: Speed

Disk drives are slow for bandwidth, but not that slow. Physical spinning drives can manage around 0.1GB/s, whereas a memory DIMM is around 10GB/s.

The big difference is on latency. Memory latencies are below 0.1μs. Disk drive latencies are at least half a revolution at typically 7,200rpm. This is about 4ms, and other factors push the number up to about 10ms. So bandwidth down by a factor of 100, but latency up by a factor of 100,000.

Accuracy

Each sector on a disk is stored with an error-correcting checksum. If the checksum detects a correctable error, the correct data are returned, the sector marked as bad, a spare 'reserved' sector used to store a new copy of the correct data, and this mapping remembered.

A modern disk drive does all of this with no intervention from the operating system. It has a few more physical sectors than it admits to, and is able to perform this trick until it runs out of spare sectors.

An uncorrectable error causes the disk to report to the OS that the sector is unreadable. The disk should never return incorrect data.

Memory in most modern desktops does no checking whatsoever. In TCM we insist on ECC memory – 1 bit error in 8 bytes corrected, 2 bit errors detected.

Data CDs use 288 bytes of checksum per 2048 byte sector, and DVDs 302 bytes per 2K sector. Hard drives do not reveal what they do.

Big Requests

To get reasonable performance from memory, one needs a transfer size of around 10GB/s $\times$ 100ns $=$ 1KB, or latency will dominate.

To get reasonable performance from disk, one needs a transfer size of around 0.1GB/s $\times$ 10ms $=$ 1MB, or latency will dominate. Hence disks have a minimum transaction size of 512 bytes – rather small, but better than one byte.

Writing a single byte to a disk drive is very bad, because it can operate only on whole sectors. So one must read the sector, wait for the disk to spin a whole revolution, and write it out again with the one byte changed. Even assuming that the heads do not need moving, this must take, on average, the time for 1.5 revolutions. So a typical 7,200rpm disk (120rps) can manage no more than 80 characters per second when addressed in this fashion.

Really

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char** argv){
  int i,fd1,fd2;
  char c;

  fd1=open(argv[1],O_CREAT|O_SYNC|O_RDWR,0644);
  fd2=open(argv[1],O_SYNC|O_RDONLY);
  write(fd1,"Maggie ",7);

  for(i=0;i<7*200;i++){
    read(fd2,&c,1);
    write(fd1,&c,1);
  }

  close(fd1); close(fd2);
  return(0);
}
```


Slow

The above code writes the string ‘Maggie ’ to a file two hundred times.

```
m1:~/C$ gcc disk_thrash.c
m1:~/C$ time ./a.out St_Margaret

real    0m13.428s
user    0m0.000s
sys     0m0.153s
m1:~/C$ ls -l St_Margaret
-rw-r--r-- 1 mjr19 users 1407 Jul 16 17:40 St_Margaret
```

So 104 characters per second. Better than we predicted, but still horrid.

Caching and Buffering

To avoid performance disasters, much caching occurs.

Applications buffer writes (an automatic feature of certain functions in the C library) in order to coalesce multiple small writes. This buffering occurs before the application contacts the kernel. If the application dies, this data will be lost, and no other application (such as `tail -f`) can see the data until the buffer is flushed. Typical default buffer sizes are 4KB for binary data, and one line for text data.

The OS kernel will also cache aggressively. It will claim it has written data to disk when it hasn't, and write it later when it has nothing better to do. It will remember recent reads and writes, and then if those data are read again, will provide the data almost instantly from its cache. Writes which have reached the kernel's cache are visible to all applications, and are protected if the writing application dies. They are lost if the kernel dies (i.e. computer crashes).

Buffering: A practical demonstration

```
$ perl -e '$|=1;for(;;){print "x";select(undef,undef,undef,0.001);}'
```

An incomprehensible perl fragment which prints the letter 'x' every millisecond.

```
$ perl -e 'for(;;){print "x";select(undef,undef,undef,0.001);}'
```

Ditto, but with buffering, so it prints 4096 x's every 4.1s.

In general output to terminals is unbuffered or line buffered, and output to files is line or block buffered, but not everything obeys this.

If a job is run under a queueing system, so that stdout is no longer a terminal but redirected to a file, buffering may happen where it did not before.

RAID

Redundant Arrays of Inexpensive/Independent Disks. These come in many flavours, and one should be aware of levels 0, 1, 5 and 6. Via hardware or software, a virtual disk consisting of several physical disks is presented to the operating system

Level 0 is not redundant. It simply uses n disks in parallel giving n times the capacity and n times the bandwidth of a single disk. Latency is unchanged, and to keep all disks equally active it is usual to store data in stripes, with a singel stripe continaing multiple contiguous blocks from each disk. To achieve full performance on single access, very large transfers are needed. Should any disk fail, all data on the whole array are lost.

Level 1 is the other extreme. It is often called mirroring, and here two disks store identical data. Should either fail, the data are read from the other. Bandwidth and latency usually unchanged, though really smart systems can try reading from both disks at once, and returning data from whichever responds faster. This trick does not work for writes.

RAID 5

RAID 5 uses n disks to store $n-1$ times as much data as a single disk, and can survive any single disk failing. Like RAID 0, it works in stripes. A single stripe now stores data on $n-1$ disks, and parity information on the final disk. Should a disk fail, its data can be recovered from the other data disks in combination with the parity disk.

Read bandwidth might be $n-1$ times that of a single disk. Write bandwidth is variable. For large writes it can be $n-1$ times that of a single disk. For small writes, it is dreadful, as, even for a full block, one has to:

read old data block and old parity block
write new data block, calculate new parity block (old parity XOR old data XOR new data), write new parity block.

Two reads and two writes, where a single disk would have needed a single write, and RAID 1 would have needed two writes, one to each disk, which could progress in parallel. RAID 5 hates small writes.

RAID 6

RAID 6 uses n disks to store $n-2$ times as much data as a single disk, and can survive any two disks failing. Colloquially refered to as 'double parity', but such an expression would offend any mathematician. Has much the same properies as RAID-5, only small writes are even worse.

RAID is not much use unless disk failures are rapidly noticed and addressed, especially in the case of RAID 5 and RAID 1 which leave no redundancy in the period between a failure occuring and the disk being replaced and refilled with the relevant data. Refilling a disk can easily take 12 hours.

RAID systems can have background 'patrol reads' in which the whole array is read, and the consistency of the data and 'parity' information checked. Such activity can be given a low priority, progressing only when the array would otherwise be idle. It can spot the 'impossible' event of a disk returning incorrect data in a block whilst claiming that the data are good.

RAID improves disk reliability and bandwidth, particularly for reads. It does little to nothing for latency.

File Systems

Disks store blocks of data, indexed by a single integer from 0 to many millions.

A file has an associated name, its length will not be an exact number of blocks, and it might not occupy a consecutive series of blocks. The filing system is responsible for:

- a concept of a 'file' as an ordered set of disk blocks.
- a way of refering to a file by a textual name.
- a way of keeping track of free space on the disk.
- a concept of subdirectories.

The data describing the files, rather than the data in the files themselves, is called metadata.

Different Solutions

Many filesystems have been invented: FAT16 and VFAT32 from DOS, NTFS from Windows, UFS (and many, many relations, such as ext2) from Unix, HFS from MacOS, and many others. They differ in:

Maximum file length.
Maximum file name length.
Maximum volume size.
Which characters are permitted in file names.
Whether ownership is recorded, and how.
Which of creation time, modification time, and last access time exist.
Whether flags such as read-only, execute and hidden exist.

Lowest Common Denominator

DOS. Supported by almost everything. Even in its VFAT32 form very slow for volumes of more than about 500GB. Maximum file size 4GB. The characters : and \ are not allowed in filenames. The old FAT16 standard of DOS is free of any patents, and mostly useless. Microsoft claims four of its patents protect the VFAT extension which allows filenames of more than 8.3 characters, and tries to charge royalties on devices using it.

NFTS, ext3 and HFS both readily support 1TB+ volumes on their own OSes. Getting them to work away from their host OSes is hard.

The UNIX Filesystem

Every UNIX vendor has one (or more) file systems of his own. However, the traditional UNIX file system (UFS) has the following features.

The UNIX file system has three types of metadata: the block bitmap, the index node (inode) and the directory entry.

The block bitmap simply contains one bit for each cluster (block) on the disk, and marks whether the cluster is free.

The directory entry is also simple: a variable-length field containing the name, and a field giving an index into the inode table.

The original UNIX filesystem was even simpler, with fixed-length 16 byte directory entries containing a 14 character name and a two byte i-node number.

Again every subdirectory contains explicit entries for '.' and '..' giving its own and its parent's inode number.

The inode table

The inode table is of fixed size, containing a fixed number of fixed-length records (typically 128 bytes each), each describing one file. Each record contains:

File length
File ownership (user and group)
File 'creation', modification and last access times
File access permissions
The number of directory entries pointing at this file
A list of the first ten clusters occupied by the file
Three pointers to clusters containing details of further clusters used

Again, the block bitmap, inode table and directory entries must all be consistent.

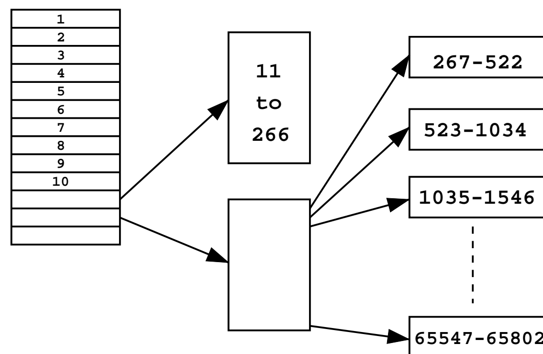
The file and group ownership records the numeric user id (typically 32 bits), not the eight character textual user name.

The program `fsck` checks for consistency. `fsck` = File System ChecK.

Large Files

Files smaller than c.10 blocks have the complete list of blocks used in their inode. Longer files use an entry which points to a disk block filled with a list of the next blocks used. If a block number is 4 bytes long, and a block is 1K, this gives another 256 blocks.

For larger files, the inode has another entry pointing to a block filled with entries pointing to blocks containing the rest of the list! This adds another 65000 or so blocks. There may be one further level of indirection for very large files.



An Experiment

```
$ dd if=/dev/zero of=test bs=1024 count=40
$ ls -l test
-rw-r--r-- 1 spqr1 users 40960 Sep  9 13:56 test
$ du -k test
40      test
```

A 40KB file takes 40KB of disk space.

```
$ ls -l test
-rw-r--r-- 1 spqr1 users 41984 Sep  9 13:58 test
$ du -k test
44      test
```

A 41KB file takes 44KB of disk space – probably using 4KB blocks.

```
$ ls -l test
-rw-r--r-- 1 spqr1 users 50176 Sep  9 14:00 test
$ du -k test
56      test
```

A 49KB file takes 56KB of disk space – thirteen 4KB blocks for the data, and one extra block for the indirect block list. (This example should be repeatable with ext3, probably not ext4.)

Fast or Slow?

Simply getting a list of files from a directory is fast – the directory needs to be read, the names are sorted alphabetically, and that is it. With luck the directory is stored in a very small number of contiguous blocks on disk.

The output of `ls -l`, `ls -F`, or a 'colour' `ls`, is much more complicated. For each entry in the directory, the corresponding inode must be opened and read in order to find out information about the file type, length, etc. Chasing a bunch of inodes, which may be widely separated on disk, is slow. Hope that they are in a cache, and there is no need to search the physical disk! (Linux's ext4 can add the file type to the directory entry, to the benefit of `ls -F`.)

The traditional directory also becomes inefficient once there are more than a thousand or so files in a directory – the traditional directory is an unindexed, variable record length, flat file database, the worst sort. More recent UNIXes use slightly saner structures, but it is generally bad to attempt to turn a filesystem into a database with tens of thousands of entries per directory.

If you do think that tens of thousands of files in a directory is a good idea, make sure that you understand that commands such as '`rm *dat`' will behave oddly if '`*dat`' expands to more than about 100,000 characters...

Open, Move, Delete

Opening a file is done by name, and involves a directory look-up to find an inode number. Once a file is open, the directory entry is irrelevant.

The `mv` command renames a file, possibly between directories, without changing its inode number.

Truncating a file to zero length and writing it out again (presumably modified) does not change its inode number. Deleting it and creating a new file with the same name does.

Deleting a file which is open does not remove it from disk – the processes which have it open will be unaware of its deletion, and only when the last process closes it will it be removed from disk.

If `mv` is used to move a file between devices, then of course the inode must change, and it is equivalent to `cp` followed by `rm`.

Hard Links

So far we have seen two file types: ordinary files and directories. UNIX also has two forms of link.

The first, the hard link, is not a new file type at all. One merely has two directory entries pointing at the same inode. As the inode stores the information about file length and access times, there are no consistency problems.

This would not be the case for DOS's FAT filesystem, in which file length and modification time are stored along with the name in the directory entry.

The link count in the inode keeps track of how many directory entries point to that inode, and only when deletion reduces the count to zero are the inode and data blocks actually freed.

All directory entries pointing at the same inode are equivalent, and must reside on the same filesystem.

Deletion (the freeing of data blocks) will also not occur if any program has the file open, even if there is no remaining directory entry pointing to its inode.

Hard links to directories are not permitted, as they would cause the directory to have multiple equivalent parents.

Hard Link Surprises

If `foo` and `bar` are hard links to each other, and hence indistinguishable, then

```
mv bar baz
```

leaves `foo` and `baz` as indistinguishable hard links. Similarly truncating `foo` and writing new contents into it leaves `foo` and `bar` identical.

However, deleting `foo` and recreating a file with the same name will break the hard link, and `foo` and `bar` will now be completely distinct, as is also the case after

```
cp bar baz; rm bar
```

which leaves `foo` and `baz` as independent files with separate inodes.

A compiler should delete and recreate its output file. Then, if its output file (i.e. an executable) is open (i.e. is being run), the run will continue uninterrupted. If it truncates and rewrites, the running program will suffer modification whilst it runs, and will crash.

An editor should truncate and rewrite, otherwise it will break hard links, which presumably existed for a reason.

A directory has a link count of two plus the number of subdirectories it has. (Consider its own '.' entry, and its subdirectories' '..' entries.) A directory with a link count of just two has no subdirectories.

Symbolic links

A symbolic, or soft, link is a new file type. The file simply contains an indirection, saying 'don't look at me, look over there instead.'

```
tcm30:/usr/sbin> ls -l /usr/sbin/sendmail
lrwxrwxrwx  1 root  system  24 Sep 12 1998
      sendmail -> /usr/local/exim/bin/exim
```

References to `/usr/sbin/sendmail` will be redirected to `/usr/local/exim/bin/exim`. The `l` at the start of the permissions bits shows that this is a symbolic link. The rest of the permissions are ignored (the file is *not* world-writable!).

The length, 24 bytes, is the number of characters in the target filename, which is stored as the file 'data'. The link and its target are quite distinguishable, and need not be on the same filesystem. The target need not even exist!

Soft links to directories are permitted. UNIX will check for circular paths in symlinks.

Soft links are usually safer, in that they lead to fewer surprises. However, the UNIX `ln` command defaults to making hard links, and one needs to specify `ln -s` for the soft sort.

Many versions of UFS, including ext2, are able to store short soft links (under about 60 characters) in the inode in place of the data block list, so no data block is allocated.

Inconsistencies

A file system will be consistent before and after a file is deleted, but not during the deletion: the directory might be changed but the block bitmap not.

And clearly if the OS has a write-behind cache, the data on the disk need not be the same the data in the cache.

Hence it is important to tell a computer to finish all disk operations and to send all modified data from its cache to the disk before turning it off. This is called *flushing* the cache, or *syncing* the disks.

('Syncing' abbreviates 'synchronising', so is similarly pronounced.)

Any filesystem which records last access times (such as UFS) will be frequently modifying data on disk.

UNIX systems, and some versions of Windows, will detect if they have been turned off without being shutdown properly, and check their disks for consistency when they are next turned on. If they have been shutdown correctly, they don't bother.

Though `fsck` and `scandisk` can often autorepair a filesystem to a consistent state, it is worth pointing out that consistency and correctness are different: formatting a disk also reduces its filesystem to a consistent state, but in a slightly unhelpful manner.

Journalling filesystems

Because checking filesystem consistency is painful on large fileservers – it can often take over an hour – various filesystems which never need a full consistency check have been developed.

They all work by keeping a log, or journal, of operations which they are about to do. Deleting a UNIX file might be broken down as:

```
write to journal 'I am about to remove this
  directory entry, free this inode, and mark
  these clusters as free.'
do the above
remove the journal entry
```

After a crash, the journal is scanned and those entries which have not been completed are finished.

A journalling filesystem must flush the journal from cache to disk before attempting the updates described by the journal.

Digital UNIX has AdvFS as a journalled filesystem, Irix has xfs, AIX has jfs, Linux has ext3, and WinNT has NTFS.

Journal problems

Journalling produces a significant performance penalty, as every write is turned into two: one to the journal, and one to the real file. For this reason most journalled filesystems only journal metadata.

Journalling metadata can ensure that the filesystem remains consistent, and guards against the type of errors which can cause whole directories to vanish. The contents of files can still be corrupted by crashes.

Journalling data as well as metadata is a serious performance penalty, and requires a much bigger area for the journal. Many journalling filesystems do not support data journalling at all.

The final problem with journalling is that hardware errors or bugs in the OS can still cause a journalled filesystem to become inconsistent. Because the recovery tools for journalled filesystems are used less frequently, they tend to be less tested and less effective.

Linux's ext3 and Solaris's UFS support journalling and still use the same layout as the older, non-journalled, filesystem they are based on. Hence the old recovery tools are valid.

Multiple programs

What happens when two programs try to manipulate the same file? Chaos, often.

As an example, consider a password file, and suppose two users change their entries ‘simultaneously.’ As the entries need not be the same size as before, the following might happen:

User A reads in password file, changes his entry in his copy in memory, deletes the old file, and starts writing out the new file.

Before A has finished, user B reads in the password file, changes his entry in memory, deletes the old, and writes out the new.

It is quite possible that A was part way through writing out the file when B started reading it in, and that B hit the end of file marker before A had finished writing out the complete file. Hence B read a truncated version of the file, changed his entry, and wrote out that truncated version.

Locking

The above scenario is rather too probable. It is unlikely that one can write out more than a few 10s of KB before there is a strong chance that your process will lose its scheduling slot to some other process.

UNIX tacked on the concept of file locking to its filing systems. A ‘lock’ is a note to the kernel (nothing is recorded on disk) to say that a process requests exclusive access to a file. It will not be granted if another process has already locked that file.

Because locking got tacked on later, it is a little unreliable, with two different interfaces (`flock` and `fcntl`), and a very poor reputation when applied to remote filesystems over NFS.

As the lock is recorded in the kernel, should a process holding a lock die, the lock is reliably cleared, in the same way that memory is released, etc.

Microsoft, trying to be positive, refers to ‘file sharing’ not ‘file locking.’

Multiple Appending

What happens when multiple programs open the same file, e.g. a log file, and later try to append to it?

Suppose two programs try to write 'Hello from A' and 'Hello from B' respectively.

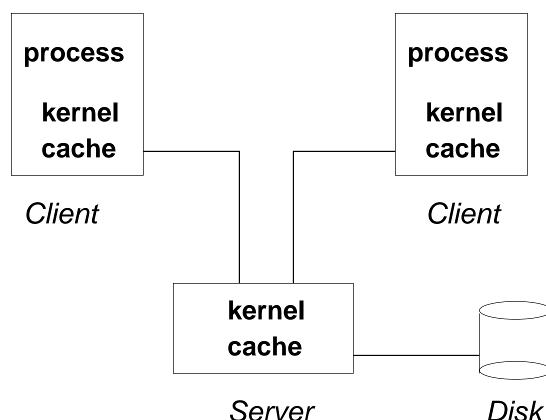
The output could occur in either order, be interleaved:
Hello frHello from A
om B
or the last one to write might over-write the previous output, and thus one sees only a single line.

The obvious problem is that the file can grow (or shrink) after program A has opened it, but before it writes to it, without the change being caused by program A.

This situation is common with parallel computers, when multiple nodes attempt to write to the same file. A set of standards called 'POSIX' states that over-writing will not occur when appending, but not all computers obey this part of POSIX.

File Servers

Filesystems are tolerably fast and reliable when the process accessing them is running on the same computer that the physical disks are in. There is one kernel to which the processes send their requests, and which controls all accesses to the disk drives. Caching reads is particularly easy, as the kernel knows that nothing apart from itself can change the contents of the disk. If remote fileservers are involved, this gets complicated. The kernel on the remote server can cache agressively, but the kernel on the machine the program is running on cannot.



Solutions

The clients could abandon all caching, and rely on the server to cache. However, this introduces a large overhead – the fastest one could hope to send a packet over a local network and get a response is about 100μs, or about 10^5 clock cycles of the CPU.

So in practice the clients do cache, and do not even always check with the server to see if their cached data are now incorrect. However, the clients dare not cache writes ever.

This restores half-tolerable performance, at the cost of sometimes showing inconsistencies.

UNIX's remote filesystem protocol, NFS, is surprisingly paranoid. The specification states that the server may not acknowledge a write request until the data has reached permanent storage. Many servers lie.

Does it Matter

If one is reading and writing large amounts of data which would not have been cacheable anyway, this is not much of an issue.

The other extreme is writing a small file, reading it in again, and deleting it. This is almost precisely what a compiler does. (It writes an object file, which is then read by the linker to produce the executable, and the object file is often deleted. It may even write an assembler file and then read that in to produce the object file.)

If this is aimed at a local disk, a good OS will cache so well that the file which is deleted is never actually written. If a remote disk is involved, the writes must go across the network, and this will be much slower.

Compiling on a local scratch disk can be *much* faster than compiling on a network drive.

On remote drives the difference in performance between `ls` and `ls -l` (or coloured `ls`) can be quite noticeable – one needs to open an inode for every file, the other does not.

Remote Performance: A Practical Example

```
$ time tar -xvf Castep.tgz
```

Scratch disk, 0.6s; home, 101s. Data: 11MB read, 55MB written in 1,200 files.

Untarring on the fileserver directly completed in 8.2s. So most of the problem is not that the server is slow and ancient, but that the overheads of the filesystem being remote are crippling. A modern fileserver on an 1GBit/s link managed 7.5s, still over ten times slower than the local disk, and over ten times slower than the theoretical 120MB/s of the server.

More crippling is the overhead of honesty. The 'Maggie' test at the start of this talk took 0.4s on the modern fileserver – impossibly fast. On the old and honest one, 20.7s.

```
$ ./compile-6.1 CASTEP-6.1.1
```

(An unoptimised, fast compile.) Scratch disk, 136s; home directory, 161s.

```
$ time rm -r cc
```

Deleting directory used for this. Scratch disk, 0.05s; home directory, 14s.

Remote Locking

The performance problems on a remote disk are nothing compared to the locking problems. Recall that locks are taken out be processes, and then returned preferably explicitly, and otherwise when the file is closed, or when the process exits for any reason. There is no concept of asking a process which has a lock whether it really still needs it, or even of requiring it to demonstrate that it is still alive.

With remote servers this is a disaster. The lock must exist on the server, so that it effects all clients. But the server has no way of telling when a process on a remote client exits. It is possible that the remote kernel, or, more likely, some daemon on the remote client, may tell it, but this cannot be reliable. In particular, if the client machine's kernel crashes, then it cannot tell any remote server that locks are no longer relevant – it is dead.

Are You There?

Networks are unreliable. They lose individual packets (a minor issue which most protocols cope with), and sometimes they go down completely for seconds, minutes, or hours (sometimes because a Human has just unplugged a cable). A server has no way of telling if a client machine has died, or if there is a network fault.

Most networked filing systems are quite good at automatically resuming once the network returns. But locking presents a problem. Can a server ever decide that a client which appears to have died no longer requires a lock? If the client has really died, this is fine. If the client is alive, and the network is about to be restored, there is no mechanism for telling a process that the lock it thought it had has been rescinded.

Similarly a client cannot tell the difference between a network glitch and a server rebooting. It expects its locks to be maintained across both events, especially because it might not have noticed either – network glitches and server reboots are obvious only if one is actively attempting to use the network or server.

Whose Lock is it Anyway

UNIX's locking mechanisms is particularly deficient. The only way of testing whether a file is locked is to attempt to lock it yourself. If you succeed, it wasn't. There is no standard mechanism for listing all locks in existance, or even for listing all locks on a given file.

Most UNIXes provide some backdoor for reading the relevant kernel data structure. This may be accessible to root only. In the case of remote locks, they will all be owned by one of the server's local NFS daemons. This makes tracing things hard. With luck the server's NFS lock daemon will provide a mechanism for listing which clients currently have locks. Even then, it will not actually know the process ID on the remote machine, as all requests will have been channelled through a single NFS daemon on the remote client.

Daemon – a long-running background process dedicated to some small, specific task.

In Linux locks are usually listed in `/proc/locks`, which is world-readable.

Breaking Locks

The safest recipe is probably as follows.

Copy the locked file to a new file in the same directory, resulting in two files of identical contents, but different inode numbers, only the original being locked.

Move the new version onto the old. This will be an atomic form of delete followed by rename. The old name is now associated with the new, unlocked file. The old file has no name, so no new process can accidentally access it.

Unfortunately the old, locked, file still exists – neither its inode nor its disk blocks are freed. If it is locked, it must be open by some process. If it is open, it cannot be removed from the disk merely because another process has removed its directory entry. Any processes which have it open will write to the original if they attempt any updates, and such updates will be lost when the last process closes the file.

Parallel Computers

Not Parallel: Multitasking

A single CPU can run only one program at once. Multitasking is an illusion for the confusion of gullible humans.

The processor runs one program for a *timeslice*, typically 1 to 20ms, then switches to another. The shorter the timeslice, the less humans will notice.

When the CPU performs a *process switch*, it must save to memory all its registers and reload the set relevant to the new process. This will take hundreds of clock cycles. The restarted process will also find the caches mostly, or entirely, storing data relevant to the previous process.

The program does not know that it had been suspended and later restarted, and needs no special code. Any I/O arriving for it will be held by the kernel and for delivery whilst the program is active.

The more registers a CPU has, the more expensive a process switch is, as more state needs to be written to memory and read back. The longer the timeslice, the less time is wasted switching.

Caches: flushed or shared?

Once a CPU is running multiple tasks via multitasking, its caches are effectively shared in some fashion. A task when switched in will find small caches are empty (i.e. full of data relating to the previous task), and large caches somewhat shared (i.e. still containing data from the process's previous scheduling slot, as well as data from the other process(es)).

How much this matters depends enormously on the applications concerned. Filling a large last level cache can be expensive – an 8MB cache will take 0.5ms to fill at an optimistic 16GB/s. A typical scheduling timeslot is around 1ms, so, by the time one has filled the cache, the scheduling slot is lost, and one never gets to reuse any of the data!

Similar arguments can be made for TLBs.

Smooth interactive performance requires short time slots, whereas efficient use of caches can require long timeslots.

Inequality

If the operating system knows a process is waiting for input (disk, network, human), it will not give that process any timeslices until input is ready for it. Such a process will be marked as *waiting* rather than *running*. The arrival of input might cause an immediate process switch to be triggered, with the timeslice of whatever process was running being interrupted. Thus fast response to I/O events is achieved.

The part of the operating system responsible for assigning priorities to processes is called the *scheduler*. The priorities need not be equal.

The UNIX `ps` command shows processes waiting for input in a state of 'wait' or 'sleep'. Only those in a state of 'run' are actively competing for CPU cycles.

The *load* or *load average* is UNIX's term for the number of proceses is the 'run' state averaged over a short period. The `uptime` command reports three averages, over 1, 5 and 15 minutes on most UNIXes.

Under UNIX the `nice` and `renice` commands can be used to decrease the scheduling priority of any process you own. The priority cannot be increased again, unless one is root. (If you use `tcsh` or `csh` as your shell, `nice` is a shell built-in and is documented in the shell man page. Otherwise, it is `/usr/bin/nice` and documented by `man nice` in the usual way.)

Parallel Computers: the Concepts

Modern supercomputers are generally *parallel computers*. That is, they have more than one CPU. So are desktops and laptops now that almost all processors have multiple cores.

Some tasks are clearly suited to being done by a 'farm' of 'workers' working simultaneously, whilst others are not. As two examples:

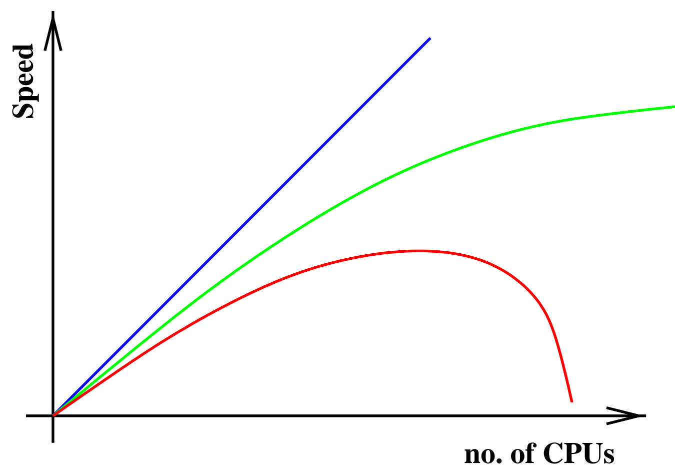
Integration of differential equation over very many timesteps. Clearly one cannot start the 5,000th timestep until the 4,999th has been finished. The process is fundamentally serial.

Dumb Factorisation of a large number. The independent trial factors from 2 to $\sqrt{n}$ are readily distributed amongst multiple processors.

A simple example of parallelisation has alrady been seen in the various 'multimedia' instructions. This is known as SIMD parallelism: Single Instruction Multiple Data. The parallelism discussed in this section is MIMD (Multiple. . .).

Scaling

How much faster does a code run when spread over more CPUs?



From top to bottom: Linear scaling (rare!), Amdahl's Law (see below), The Real World

Notice that the speed is not monotonic in the number of CPUs

Amdahl's Law

Amdahl was a pioneer of supercomputing and an employee of IBM.

This law assume that a program splits neatly into an unparallelisable part, and a completely parallelisable part. It claims:

$$t_n = t_s + t_p/n$$

The total run time on n processors is the time for the serial part of the code, plus the time the parallel part would take on a single processor divided by the number of processors.

Consider $t_s = 0.2$ and $t_p = 0.8$. Then $t_1 = 1.0$, $t_{32} = 0.225$ and $t_\infty = 0.2$.

On 32 processors the speedup is $4.5\times$ and the efficiency is just 14%.

Bigger is better

Suppose t_s and t_p scale differently with problem size.

Assume t_s scales as N and t_p as N^3 and consider a problem $4\times$ as large as before. Now

$t_s = 0.8$ and $t_p = 51.2$ giving $t_1 = 52$ and $t_{32} = 2.4$.

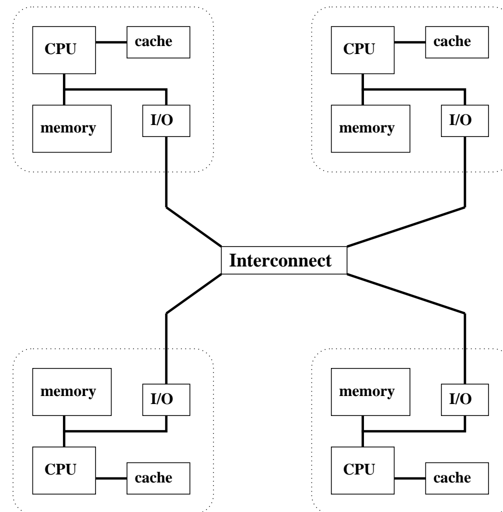
Now the speedup on 32 processors is $21\times$, and the efficiency is now over 67%.

Supercomputers like *big* problems.

Conversely, workstations hate big problems, as their various caches become less effective and their overall efficiency falls.

MPP and SMP

We first consider the MPP design of parallel computer. It is simple, consisting of lots of separate single-processor computers with a fast network between them. Each separate sub-computer, or node, has its own memory, and, in some cases, even its own disk drive. Such a parallel computer is called a *distributed memory computer* or *massively parallel processor*



Breaking the Code

This arrangement is so far removed from the traditional model of a computer, that traditional code does not run on it. The programmer must be prepared to think in terms of multiple processors working on his program at once, each with its own private memory, and any interprocessor communication being explicitly requested.

Fortunately this is not nearly as hard as it might sound, and there are standard programming models to assist. Thus one can write code for a Cray T3E, using C or FORTRAN with MPI, and be confident that it will run, unmodified, on an IBM SP, a Beowulf cluster, or on a machine not yet developed. One merely has to follow the relevant standards and not be lured down the road of vendor-specific extensions…

MPI (1994) and PVM (1991, now obsolete) standardised the programming model for MPPs. Before PVM, each vendor had its own way of doing things.

Topologies

There are many different ways of connecting nodes together, as ever governed by cost and practicality.

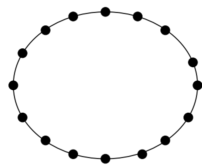
Two useful ways of characterising a network are the 'diameter', the maximum number of hops from one node to another, and the bisectional bandwidth, the bandwidth between two halves of the machine.

	Bandwidth	Diameter
Ring	2	$N/2$
2D Grid	$\sqrt{N}$	$2\sqrt{N}$
2D Torus	$2\sqrt{N}$	$\sqrt{N}$
Hypercube	$N/2$	$\log_2 N$
Tree	2	$2\log_2 N$
Fat tree	$N/2$	$2\log_2 N$
X-bar	$N/2$	1
3D X-bar	$N/2$	3

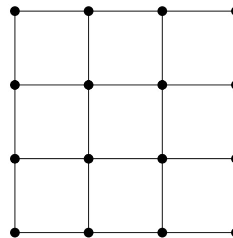
The Cray T3D was a 2D torus, the IBM SP2 a fat tree, the SGI Origin2000 a form of hypercube, and the Hitachi SR2201 a 3D X-bar.

Ideally the network topology should not be apparent to the user.

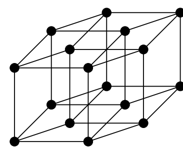
16 Nodes...



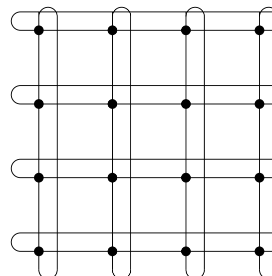
Ring (1D torus)



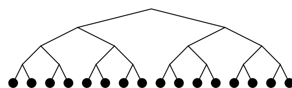
2D mesh



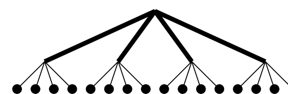
Hypercube



2D torus



Tree (log 2)



Fat Tree (log 4)

Performance

Another important characteristic of the interconnect is its raw performance, both bandwidth and latency. These are most usefully measured using a standard interface such as MPI, and not using the hardware directly.

Ideally the time to transmit a packet is simply

latency + size / bandwidth

If size $<$ latency $\times$ bandwidth, then the latency will dominate.

Also ideally communication between a pair of nodes is unaffected by any other communications happening simultaneously between other nodes. Such a network is called *non-blocking*.

Typical figures are 1 to 3 GB/s bandwidth and 1 to 3 μs latency. Clusters using 1GBit/s ethernet typically run at around 100 MB/s and 20 μs.

Parallelisation Overheads

Amdahl's law assume that there are no overheads associated with parallelisation. This is certainly a gross approximation.

Consider the case where each node must exchange data with every other node at some point in the program: some sort of rearranging of an array spread over all the nodes. E.g. an FFT

Each node must send $n-1$ messages of size a/n where a is the size of the distributed array. Even assuming that the nodes can do this simultaneously, the time taken will be

$$(n-1) \times \left(\lambda + \frac{a}{n\sigma}\right) \approx n\lambda + \frac{a}{\sigma}$$

where λ is the latency and σ the bandwidth.

Amdahl revisited

A better form of Amdahl's law might be

$$t_n = t'_s + t_p/n + c\lambda n$$

where $t'_s > t_s$.

Now t_n is no longer a monotonically decreasing function, and its minimum value is governed by λ.

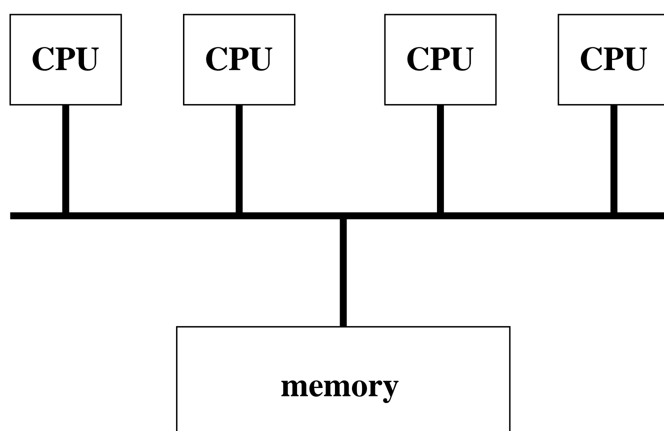
This form stresses that the quality of the interconnect can be more important than the quality of the processors.

Hence 'cheap' PC clusters work well up to about 16 nodes, and then their high latency compared to 'real' MPPs starts to be significant.

SMP: Bused Based

SMP (Symmetric Multi Processor, Shared Memory Processor) describes another class of multi-CPU computer.

The original, bus-based, SMP computer simply has multiple CPUs attached to a single system bus.



The architecture is *symmetric* (all CPUs are equivalent), and the memory is *shared* between them.

Shared memory

This is precisely what a modern multi-core CPU looks like, as a single core is equivalent to the old idea of a CPU.

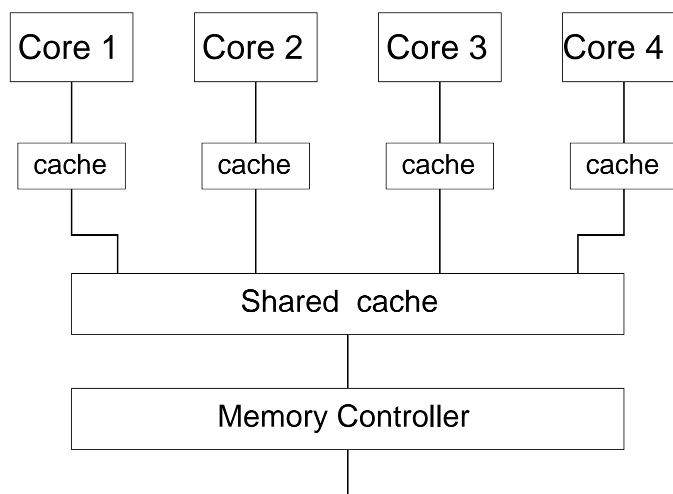
As all processors access the same main memory, it is easy for different parts of a program executing on different processors to exchange data. One CPU can write an array into memory, possibly from disk, possibly as the result of a calculation, then all other CPUs can read it with no further effort.

Programming is thus simple: all the data are in one place, and there is merely the little matter of dividing up the millions of instructions to be executed in a long loop between the multiple eager processors – a job so simple that the compiler can do it automatically.

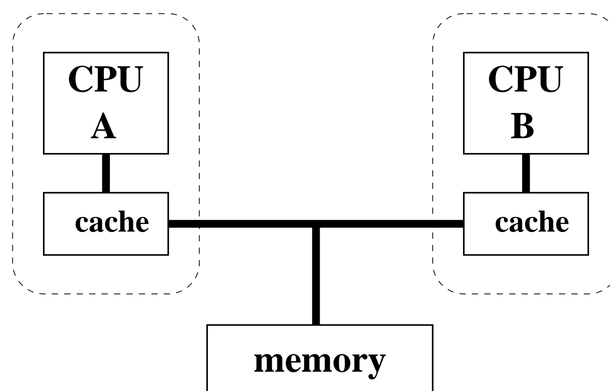
Except it is not quite that simple.

Two Heads are Better than One?

As in a conventional, single-CPU computer, the single processor typically spends between 75 and 95% of its time waiting for memory, trying to 'feed' two or more CPUs from one memory bank is clearly crazy. The memory was, and is, the bottleneck. The CPU was not. However the design is cheap, and is now ubiquitous within a CPU, as the diagram below illustrates, and is common in multi-socket designs.



Cache coherency



Processor A reads a variable from memory. Later, it reads the same variable, which it can now get directly from its cache, without troubling the system bus.

Only it can't. For what if processor B has modified that variable, and processor A needs the new value? If processor B has a write back cache, the new value may not even have reached the main memory, with the current value being held in processor B's cache only.

Snoopy caches

The trivial solution is to abandon all caches.

An easy solution is to ban write-back caches, and to ensure that each cache '*snoops*' the traffic on the system bus, and, if it sees a write to a line it is currently caching, it must either update itself automatically, or mark its copy as being invalid.

These solutions severely compromise one's cache architecture, and often lead to a SMP machine generating more traffic to the main memory than a uniprocessor machine would running the same code. Thus a SMP machine can fail to reach the performance of a single-processor workstation based on the same CPU.

With either of these solutions, the definitive data are always those in the main memory.

Even single core single CPU workstations have a lesser version of this problem, as it is common for the CPU *and* the disk controller to be able to read and write directly to the main memory. However, with just two combatants, the problem is fairly easily resolved.

More Complexity: NUMA

Most modern SMP machines are not bus based. Internally they are configured like MPPs, with the memory physically distributed amongst the processors. Much magic makes this distributed memory appear to be global.

This (partially) addresses the poor memory bandwidth of the bus based SMP machines.

However, there are problems. . .

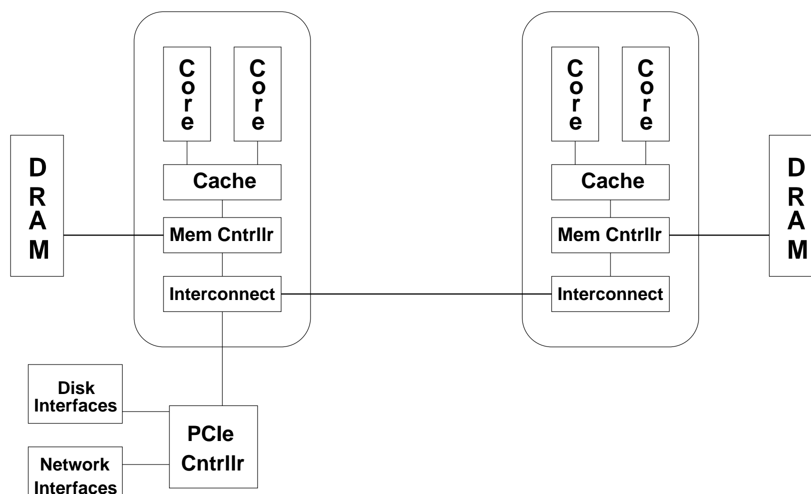
Not least that some memory is now local to a CPU, some attached to its neighbour(s), some its next nearest neighbour(s). Access times and bandwidths will vary depending on the relationship between the physical location of the memory and the core on which a process is executing. Hence the acronym Non Uniform Memory Architecture.

And magic costs money, and, in this case tends to degrade performance over an MPP, providing instead increased flexibility.

To emphasise that majic has been used to make even caches work correctly, one sometimes sees the acronym cc-NUMA, cache-coherent NUMA. Fortunately the alternative of non cache coherent NUMA is extremely rare.

Modern, small SMPs

Modern processors not only contain multiple cores, but also often contain the interconnect needed to construct SMP machines of two or four sockets. (The below diagram should really show four or six cores per CPU.)



AMD's HyperTransport (HT) interconnect and Intel's rather later Quick Path Interconnect (QPI) are quite similar. In both cases a CPU has about three links, which can connect either to other CPUs, or to I/O controllers (e.g. PCIe bus controllers). The above diagram shows the left-hand processor using two links, and the right-hand one. Not only is the memory NUMA, but so is access to disk and network.

NUMA in Action

A version of the Streams benchmark written in MPI gives a measure of memory bandwidth. The command `taskset` can be used to specify which cores it runs on. Here a dual socket machine with quad core processors:

Process count	cores used	bandwidth
1	1	9.8 GB/s
2	1&3	12.3 GB/s
2	1&5	12.3 GB/s
2	1&2	19.6 GB/s
4	1,3,5&7	13.0 GB/s
4	1,2,3&4	24.1 GB/s
4	1,2,5&6	24.1 GB/s
8	1–8	26.0 GB/s

Conclusion: each core has maximum b/w of c.10 GB/s
each socket has maximum b/w of c.13 GB/s

This was a dual socket 2.4GHz quad core Intel 'Nehalem'. Each processor has a 24 byte wide bus to DDR3/800 memory, so theoretically 19.2GB/s per socket. It seems that cores 1, 3, 5 & 7 are on one CPU (socket), and the even numbers on the other. Note that here the performance gain in moving from one process to eight on an eight core machine for a code with no inter-process communication is a factor of merely 2.7. The gain from using one core per socket to all four is a factor of just 1.3.

The Consequences of NUMA

If a processor is mangling an array, it now matters crucially that that array is stored in the memory on directly attached to that processor, and not on memory the other side of the machine. Getting this wrong can drop the performance by a factor of three or more instantly.

Whereas with MPP all memory accesses are guaranteed to be local, as one cannot access remote memory except by explicit requests at the program level, with SMP the compiler has many ways of getting things wrong.

```
for(i=0;i<10000000;i++)
   t+=x[i]*y[i];
```

Consider this on a two processor NUMA machine. If the code is split so that each processor stores the first 5000000 elements of each array in its directly attached memory, and does the first half of the loop, then optimal performance is obtained. If the whole of `x` is stored in the memory local to one processor, the whole of `y` the other, then much reduced performance will result.

MESI solutions for caches

A typical SMP has extra bits associated with each cache line, which mark it as being on one of four states:

- Modified (i.e. dirty) – this state exists for uniprocessor machines too
- Exclusive (in no other cache)
- Shared (possibly in other caches too)
- Invalid

Modified implies exclusive, and a line must be exclusive before it can be modified.

A line fill for a read ensures that no other cache has the line modified or exclusive, then loads the line marked as 'shared.' A fill for a write also ensures than any caches with that line shared mark it invalid. In either case any cache with it 'modified' (there can be only one) writes it back to memory.

Thus a line can be:
In no caches
In one cache and marked as modified (or exclusive)
In one or more caches and modified (or exclusive) in none

More Messes

It may seem as though ensuring that each thread works on its own data, and rarely exchanges data with other threads, is sufficient to ensure performance. It isn't, for there is the overhead of checking to ensure that one core is not trying to update data held in another core's cache. All modern computers do this in hardware. Even if the compiler knows that two data items are distinct, the hardware will still check, and may need to do so as a thread may migrate from one core to another during its execution.

One method of checking simply broadcasts details of all line fills to all cache controllers, and the fill does not progress until the other controllers have had an opportunity to reveal that they held the line. The amount of broadcast traffic tends to scale as the square of the number of caches, so this works poorly for large numbers of CPU – in practice, beyond about four.

A significant improvement uses a 'directory'. A directory entry is associated with each line in memory, and records which caches have copies of the line. Then a fill need simply check the directory, contact only those caches listed (probably none), and proceed, updating the directory as it does so. In practice a directory which only provides partial coverage of the main memory can be used, falling back to broadcasting when the directory is incomplete.

Secondly, the important concept is not the sharing of data, but the sharing of cache lines. If two threads attempt to write to adjacent items in the same cache line, this is no better from the point of view of copying data around in a MESI system than if they were writing to the same element. This is sometimes called 'false sharing'.

Broadcast Failures

In 2006 I had the fun of testing an 8-socket Opteron server with dual core processors. The board design was cheap, and not quite up to AMD's recommendations, so the following results are not a fair reflection on AMD…

The measured bandwidth using the Streams benchmark was 2.0GB/s for a single process, peaked at 8.0GB/s for six, and 7.4GB/s for sixteen. Why am I convinced that this reflects a severe broadcast problem? The server design allowed me to remove physically four of the CPUs. The numbers I then measured were 4.4GB/s for a single process, and 16.3GB/s for eight.

Simply having the extra four CPUs present, and informed of what was happening, even if one did not use them, halved the preformance of this machine for Streams! A single process memory latency benchmark moved from a poor 190ns to a very poor 290ns just by having the extra CPUs present.

Even Linpack, normally forgiving of poor memory subsystems, managed 26.4 GFLOPS running on eight cores (four CPUs present), and only 22 GFLOPS on 16 cores (8 CPUs present) with an array size of 40,000. The cores had a theoretical peak performance of 4.4 GFLOPS each.

Sharing, True and False

Naturally things get worse if multiple processors really are trying to update the same memory location. Not only does this need detecting, but once detected, it needs action to ensure that correct behaviour is observed. Corrective action tends to involve the automatic transfer of cache lines between CPUs. Not fast, as lines are big.

However, the important concept is not the sharing of data, but the sharing of cache lines. If two threads attempt to write to adjacent items in the same cache line, this is no better from the point of view of copying data around in a MESI system than if they were writing to the same element. This is sometimes called 'false sharing'.

A False Sharing Example

```
#pragma omp parallel for private(j,ptr1,ptr2)
  for(i=0;i<=1;i++){
    if(i==0){
      ptr1=line;
      ptr2=line+2*OFFSET;
    }
    else{
      ptr1=line+OFFSET;
      ptr2=ptr1+3*OFFSET;
    }
    for(j=0;j<(1<<28);j++){
      *ptr1+=*ptr2;
    }
  }
```

The above takes 2s to execute in a serial fashion on a certain dual core machine with a particular compiler. In parallel, it takes 1s. Unless OFFSET is one or two, in which case it takes over 12s in parallel, and still 2s in serial.

Inclusive Levels

There are three common approaches to a cache hierarchy:

- Data in one level is guaranteed to be in no other level.
- Data in level n is guaranteed to be in all levels $> n$.
- Neither of the above guarantees

Intel likes the second scheme, *inclusive* caches. In response to cache coherency requests, it need only check the last level cache, for if the data are not there, they can be in no other level.

AMD likes the first, *exclusive* caches, for the total amount of data cached is then the sum of the sizes of the levels, not simply the size of the last.

Programming Example

Consider doing an enormous dot product between two arrays previously set up. The SMP code might look as follows:

```
! Let's hope the compiler optimises
! this loop properly

t=0.0
do i=1,100000000
  t=t+a(i)*b(i)
enddo
```

Easy to write, but little control over whether it is effective!

To be fair, HPF (High Performance Fortran) and OpenMP (a set of directives to Fortran and C) permit the programmer to tell an SMP compiler which sections of code to parallelise, and how to break up arrays and loops. One day I might meet someone using such a language for real research.

Programming, MPP

```
! Arrays already explicitly distributed
! Do the dot product for our bit

t_local=0.0
do i=1,nmax  ! nmax approx 100000000/ncpus
  t_local=t_local+a(i)*b(i)
enddo

! Condense results

call MPI_AllReduce(t_local,t,1,   &
   MPI_DOUBLE_PRECISION, MPI_SUM, &
   MPI_COMM_WORLD)
```

(Those MPI calls are not half as bad as they look once one is used to them!)

All the variables are local to each node, and only the MPI call causes one (`t`) to contain the sum of all the `t_local`'s and to be set to the same value on all nodes. The programmer must think in terms of multiple copies of the code running, one per node.

The Programming Differences

With MPP programming, the programmer explicitly distributes the data across the nodes and divides up the processing amongst the nodes. The programmer can readily access the total number of CPUs and adjust the distribution appropriately.

Data are moved between nodes by explicitly calling a library such as MPI.

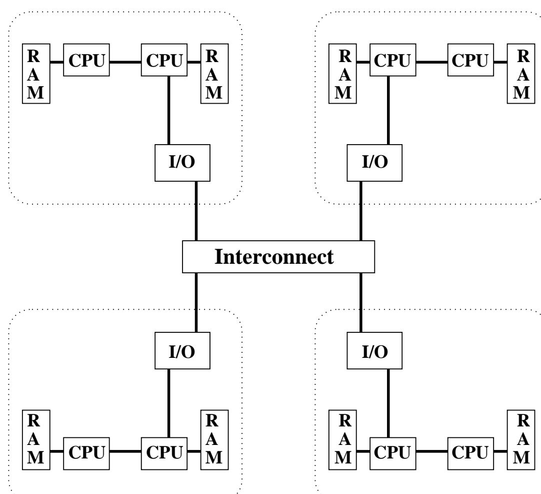
With SMP, the compiler tries to guess how best to split the task up amongst its CPUs. It must do this without a knowledge of the physical problem being modeled. It cannot know which loops are long, and which short.

Artificial intelligence vs human intelligence usually produces a clear victory for the latter!

The MPP model will work, often quite efficiently, on an SMP machine. The converse is not true.

Modern, large MPPs

Modern MPP designs join SMP nodes like the above. Such a machine is awkward to program optimally, as one has both internode and intranode parallelism to address, with two very different interconnect speeds. A program which is merely correct, but not necessarily optimal, can ignore the SMP nature of the nodes, and consider the machine to be an MPP of all the cores.



Differing Speeds

The link between a CPU and its directly-attached memory is typically a 128 bit (or 256 bit) 1666MHz DDR3 bus. Theoretical bandwidth 26GB/s (or 52BG/s), unidirectional.

The link between CPUs is typically 16 bit 6.4GT/s HyperTransport or QPI. Bandwidth 13GB/s, supporting traffic in both directions simultaneously.

The link from the I/O controller to the interconnect controller is typically 16 bit PCIe 2.0. This has a theoretical bandwidth of 8GB/s, and is bidirectional.

The interconnect itself is often quad data rate Infiniband 4X. This has a theoretical bandwidth of 4GB/s, and is bidirectional.

Measured latencies vary more widely, from under 0.1μs for accessing memory within a node, to just over 1μs for an MPI transfer between nodes.

Infiniband is not the only possible inter-node interconnect. It is the most common of the specialised interconnects, other examples including CrayLink and Myrinet.

The cheap and nasty option is simply to use 1GBit/s ethernet, for a theoretical bandwidth of 0.1GB/s and a latency of around 20μs.

(All data current in 2013.)

Does Speed Matter?

The current trend seems to be for nodes to be increasing in internal bandwidth, and in peak MFLOPS, rather faster than (cheap) inter-node interconnects are increasing in speed. The first MPP I used seriously, a Hitachi SR2201 installed in 1997, had a per node performance of 300MFLOPS, and an interconnect peak bandwidth of 300MB/s, so one byte of interconnect bandwidth per FP operation.

A current node would probably have two processors, each of eight cores running at 2.5GHz and capable of eight FP ops per clock cycle. With QDR Infiniband 4X this is one byte of interconnect bandwidth per 80 FP operations.

For some algorithms this is still so much more than is needed that one need not care. For others, careful division of one's code to reflect the different performance of intra-node and inter-node transfers can be beneficial (if time-consuming). One approach is to use OpenMP within a node, and MPI between nodes. Another is to attempt to get topology information into the MPI system.

Strong or Weak?

Much is said about overlapping communication and computation in MPI codes. This assumes that separate hardware is responsible for communication, and is called 'strong progress.'

In practice, most MPI implementations rely on the CPU for both communications and calculations. Worse, MPI transfers can be progressed only whilst MPI calls are being made. This results in 'weak progress.'

Process 0	Process 1
Large, non-blocking send	Matching blocking receive
Lots of computation	[no data transferred]
Another MPI call	[more data transferred]

In Practice

```
#include <stdio.h>
#include <mpi.h>
#include <time.h>
#define SIZE 4096

int main(int argc, char *argv[]){
  int this_process, num_processes,*a;
  MPI_Status status;
  MPI_Request req;
  time_t t1;

  a=malloc(SIZE);
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&num_processes);
  MPI_Comm_rank(MPI_COMM_WORLD,&this_process);
  printf("Process %d out of %d\n",this_process,num_processes);

  MPI_Barrier(MPI_COMM_WORLD);
  t1=time(NULL);

  if (this_process==0){
    MPI_Isend(a,SIZE/sizeof(int),MPI_INT,1,123,MPI_COMM_WORLD,&req);
    sleep(10);
  }
  else if (this_process==1){
    MPI_Recv(a,SIZE/sizeof(int),MPI_INT,0,123,MPI_COMM_WORLD,&status);
  }

  printf("Process %d reaching barrier at time %d\n",this_process,
       (int)(time(NULL)-t1));
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Results

SIZE=2048

```
Process 0 out of 2
Process 1 out of 2
Process 1 reaching barrier at time 0
Process 0 reaching barrier at time 10
```

SIZE=4096

```
Process 0 out of 2
Process 1 out of 2
Process 1 reaching barrier at time 10
Process 0 reaching barrier at time 10
```

The behaviour of code like this will vary enormously depending on the details of one's MPI implementation. If one has fewer cores than MPI processes, so the scheduler might not be able to run sending and receiving processes simultaneously, other major slow-downs might occur.

Please don't write C like the above – no error codes were checked, the excuse here being that it needed to fit on one slide!

The MPI implementation used for the above was OpenMPI 1.6.3 running on a dual core computer.

Multithreading

Whether in a uni- or multi-processor computer, the CPU is often used very inefficiently, with most of its functional units idle waiting for memory to respond or data dependencies to be resolved. It is rare for a four-way superscalar CPU to be able to issue four instructions simultaneously.

Conventional multitasking is not the answer. This software-driven process-switching takes thousands of clock cycles, so is useful for latencies caused by disk drives, networks and humans.

However, there are rarely data dependencies between processes, so in some sense multitasking is the answer.

A multithreading processor gains multiple banks of registers, one per 'thread' (process) which will be run simultaneously. These processes share access to the functional units, caches, instruction decoding logic, etc.

SMT

There are different ways of achieving multithreading. Some change thread every clock-cycle, whereas the more advanced Simultaneous MultiThreading allows instructions from different threads to be issued in the same cycle.

The extra logic on the CPU need to keep track of a modest number of threads is very small, increasing the CPU size by less than 10%. The gain is zero if the computer is only ever running a single thread, but sometimes the throughput can increase by over half when two threads are run.

However, multithreading, or hyperthreading as Intel calls it, gives one no more memory bandwidth, no more functional units, and no more caches. A processor supporting two threads generally appears to the user as two processors, but these virutal processors will share the caches, functional units and memory controller that were dedicated to a single core. For most scientific codes there is no gain to be had here.

Intel's 'Pentium4 with Hyperthreading' (2002) supported two threads per processor (core), but instructions from different threads could not be dispatched in the same clockcycle. Intel's Core and Core 2 processors abandoned hyperthreading, but the current Core i5/i7 line has reintroduced it, improved so that instructions from different threads can be dispatched on the same cycle. Processors from IBM and Sun also support two threads per core. Sun's UltraSPARC T1 series supports up to eight threads per core. AMD has yet to offer any form of hyperthreading. Gains tend to be low in tightly coupled parallel jobs – if all threads do the same thing at the same time, then usually one part of the CPU is overloaded, and the rest idle, with or without SMT. SMT works better when different threads are making different demands on the CPU.

-r8, 217
.COM file, 164
/proc, 153
0x, 60

address lines, 47, 48
allocate, 142
Alpha, 28
AMD, 24
Amdahl's law, 280, 287, 288
ARM, 24
assembler, 28
ATE, 70

bandwidth, interconnect, 286
basic block, 218
bisectional bandwidth, 284
BLAS, 83
branch, 27
branch prediction, 29
bss, 140
buffer
  IO, 242, 243
bus, 15

C, 221
cache
  anti-thrashing entry, 70
  associative, 69, 73
  direct mapped, 66
  exclusive, 303
  Harvard architecture, 74
  hierarchy, 71
  inclusive, 303
  line, 63
  LRU, 73
  memory, 56, 57, 61–292
  write back, 72–74
  write through, 72
cache coherency
  broadcast, 299, 300
  directory, 299
  snoopy, 72, 293
cache controller, 58
cache line, 298
cache thrashing, 68
caches
  and multitasking, 276
cc-NUMA, 294
CISC, 22, 26
clock, 15
compiler, 213–217
compilers, 25
cooling, 78
CPU family, 24
crossbar, 284
CSE, 199

data dependency, 21, 23
data segment, 140
debugging, 193–195, 211, 216, 217
diameter, 284
dirty bit, 72
disk thrashing, 130
distributed memory computer, 282
division
  floating point, 35
DRAM, 45–51, 56
DTLB, 128

ECC, 53–55, 74
EDO, 49
ELF, 171, 173
ext3, 260, 261

F90, 221
F90 mixed with F77, 226
false sharing, 299, 301
FAT32, 248, 249
file locking, 263, 269–272
flash RAM, 45
FPM, 49
FPU, 14
`free`, 134
`fsck`, 259
function overloading, 193
functional unit, 14, 20

heap, 140–142, 145–147
hex, 59, 60
hit rate, 57, 70
HPF, 304
hypercube, 284
hyperthreading, 314
HyperTransport, 308

in-flight instructions, 29
Infiniband, 308
inlining, 209
inode, 250–252, 258
instruction, 17, 18
instruction decoder, 14
instruction fetcher, 14
Intel, 24
issue rate, 20

ITLB, 128

journalling filesystems, 260, 261

kernel, 175

latency, functional unit, 20
latency, interconnect, 286
libraries, shared, 143
link
  hard, 256, 257
  soft, 258
linking, 180–193, 216
Linpack, 40, 300
`ln`, 258
load, 277
loop
  blocking, 94, 95, 111–114
  coda, 202
  elimination, 108, 205
  fission, 207
  fusion, 206, 231
  interchange, 212
  invariant removal, 199
  pipelining, 210
  reduction, 203
  strength reduction, 208
  unrolling, 92, 202
`ls`, 254, 267

`main()`, 183, 185
`malloc`, 142
matrix multiplication, 82, 83
MESI, 298
metadata, 247, 250

MFLOPS, 39
micro-op, 26
microcode, 34
MIPS, 24, 39
`mmap`, 141, 144–147
MPI, 283, 305, 306, 310–312
  progress, 310–312
MPP, 282, 283
multitasking, 275
multithreading, 313, 314
`mv`, 255

name mangling, 193
NFS, 266
`nice`, 277
non-blocking, 286
null pointer dereferencing, 141
NUMA, 294–297

OpenMP, 304
operating system, 130, 277
optimisation, 197
out-of-order execution, 31

page, 135–137
page colouring, 138
page fault, 130
page table, 126, 127, 129
pages, 122–124
  locked, 131
paging, 130
parallel computers, 278
parity, 52, 74
PCIe, 308
physical address, 123

physical address, 122, 124
physical memory, 132
pipeline, 18–20, 27
pipeline depth, 18
power, 78
prefetching, 75, 76, 204
priority, 277
process switch, 275
`ps`, 132, 133, 277

QPI, 308

RAID 1, 244
RAID 0, 244
RAID 5, 245
RAID 6, 246
register, 14
register spill, 101, 207
registers, 207
`renice`, 277
RISC, 22, 26, 34
`rm`, 255

scaling, 280, 281, 288
scheduler, 277
SDRAM, 49
segment, 140
segmentation fault, 124, 145, 148
sequence point, 219, 220
shared memory processor, 289
SIMD, 278
SMP, 289, 294
SMT, 314
SPEC, 41, 42
speculative execution, 30

SRAM, 45, 56
stack, 140–142, 152, 178, 195
stalls, 29
streaming, 76
Streams, 296, 300

tag, 61–67
`taskset`, 296
text segment, 140
timeslice, 275
TLB, 125, 128, 129, 135
topology, 284
trace (of matrix), 225–228
tree, 284

UFS, 248, 250
`ulimit`, 142
`uptime`, 277

vector computer, 36–38
virtual address, 122–124
virtual memory, 130, 132
voltage, 78

x87, 34